# Clube Delphi

Ano 9 :: Edição 125 :: R$ 11,90

DevMedia

ISSN 1517990-7

**Firebird 2.5**
Conheça os principais recursos do banco de dados mais utilizado pela comunidade Delphi

**Expert**
Crie plug-ins para o IDE

# CODESITE

## Log e depuração avançada no Delphi XE

**Quick Update**
Desenvolvimento para iPhone com Delphi

**DataSnap no Delphi XE**
Aplicações distribuídas baseadas em REST e JavaScript

**Boas Práticas**
Desenvolvimento Ágil e Testes Unitários

**Brinde**

**Comprando esta edição ganhe 3 vídeo-aulas:**

- Crystal Reports – Introdução
- Crystal Reports– Menu de Ferramentas
- Delphi Prism – Criando uma camada de acesso a dados

# Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

## A DevMedia simplifica pra você!

easy .net magazine

### Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

### Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

**A Easy.Net não é feita em PDF ou Flash!**
**Ela utiliza um aplicativo 100% online.**

DevMedia group

**Acesse agora a edição Nº 1 totalmente grátis e comprove**

**www.devmedia.com.br/easynet**

**Easy.Net Magazine, seu curso inicial de .Net!**

# Sumário


Quick Update

**05 – MonoDevelop e Prism**
[ *Sebastian P.R. Gingter* ]

Delphi

Banco de Dados

**08 – Firebird 2.5**
Conheça os principais recursos do banco de dados mais utilizado pela comunidade Delphi
[ *Adriano dos Santos Fernandes* ]

Boas Práticas

Boas Práticas – Novidades

**18 – DataSnap no Delphi XE**
Aplicações distribuídas baseadas em REST e JavaScript
[ *Rodrigo Carreiro Mourão* ]

Boas Práticas – Expert

**25 – Estenda seu Delphi XE**
Aprenda a criar experts e plug-ins
[ *Vitor Luiz Rubio* ]

Boas Práticas

**33 – CodeSite**
Log e depuração avançada no Delphi XE
[ *Rodrigo Araujo* ]

Boas Práticas - Engenharia de Software

**40 – Desenvolvimento ágil**
Entenda a relação entre desenvolvimento ágil e testes unitários
[ *Rafael Stavarengo* ]


Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

## Brinde na web desta edição - 3 vídeos! Confira:

3 Vídeos

- **Curso Crystal Reports XI – Introdução**
  *http://www.devmedia.com.br/post-18540-Introducao--Curso-Crystal-Reports-XI-com-Delphi-2010-Parte-1.html*

- **Curso Crystal Reports XI – Menu de Ferramentas**
  *http://www.devmedia.com.br/post-18541-Menu-de-Ferramentas--Curso-Crystal-Reports-XI-com-Delphi-2010-Parte-2.html*

- **Curso Delphi Prism – Criando uma camada de acesso a dados (DAL)**
  *http://www.devmedia.com.br/post-18344-Criando-a-camada-de-acesso-a-dados-Multicamadas-Curso-Completo-Delphi-Prism-Parte-69.html*

Para acessar os vídeos utilize os seguintes dados
*Login:* **DVM.PLS** *Senha:* **K4N8AA0**


**Gostou das vídeo aulas?** O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

## Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

*www.devmedia.com.br/clubedelphi/feedback*

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PLS** Senha: **K4N8AA0**

Clube Delphi

Ano 9 - 125ª Edição 2010 - ISSN 1517990-7 Impresso no Brasil

## Corpo Editorial

**Editor Geral**
Guinther Pauli
guinther@devmedia.com.br

**Editor Técnico**
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Laporte
daniel.sobrinho@devmedia.com.br

**Equipe Editorial**
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

**Jornalista Responsável**
Kaline Dolabella - JP24185

**Capa e Diagramação**
Romulo Araujo
romulo@devmedia.com.br

**Coordenação Geral**
Daniella Costa - daniella@devmedia.com.br

**Revisão e Supervisão**
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

**Distribuição**
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

**Atendimento ao Leitor**

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

**Isabelle Macedo e Uellen Goulart** – Atendimento ao Leitor
*www.devmedia.com.br/mancad*
(21) 3382-5038

**Kaline Dolabella –** Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

**Publicidade**

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

**Cristiany Queiróz**
publicidade@devmedia.com.br

# EDITORIAL

Esta edição continua a nossa cobertura das principais novidades do Delphi XE. Nosso destaque este mês vai para o CodeSite, a melhor ferramenta de Logging e Debug para Delphi. Mas o Delphi já não tem um excelente depurador integrado? Para que eu preciso de outro depurador? O CodeSite pode ser classificado como um eficaz sistema de Logging, ou ainda, de instrumentação de código, também podendo atuar na técnica de Profiling. Imagine que estamos desenvolvendo uma grande aplicação, com milhares de linhas de código, classes de negócio, componentes, acesso a dados, pacotes, interfaces VCL etc. É muito comum nos remetermos ao que chamamos de "depurador" dos pobres quando é preciso inspecionar o próprio comportamento da aplicação em execução. Por exemplo, o envio de informações para a tela ou para um arquivo, sobre consumo de memória, ou tempo de execução de procedimentos, a fim de detectar gargalos, verificar estados de objetos, alcance de pontos de execução do sistema e mais. Isso se torna ainda mais claro em ambientes de produção com servidores DataSnap ou Web, tipicamente aplicações multithread, onde simplesmente não temos como depurar uma aplicação em uma situação real de execução, pois: 1) não é simples depurar um servidor rodando remotamente com o IDE na máquina de desenvolvimento, a saída seria o Remote Debugger 2) é praticamente impossível debugar aplicações multithread, ainda mais se usarem eventos. O CodeSite permite que o desenvolvedor "injete" código de inspeção (vamos chamar de um ShowMessage "tunado"), onde é possível enviar mensagens, objetos e tipos personalizados para um ambiente real de log ("ao vivo"), podendo ainda ser analisado posteriormente. É aqui vai a melhor notícia, o CodeSite que antes era uma ferramenta separada, agora vem junto com o Delphi e é integrado ao IDE, como você vai conferir na matéria do Rodrigo Araujo.

Ainda nesta edição, uma matéria especial sobre o Firebird 2.5, escrita por um dos desenvolvedores do banco, o Adriano. E mais novidades, o Rodrigo Carreiro detalha o uso dos novos recursos para suporte a REST no DataSnap do Delphi XE. Criação de Experts e Plug-ins para estender o IDE do Delphi XE é o tema da matéria do Vitor Rubio. Finalizando, mais boas práticas, com uma matéria sobre Testes do Rafael Stavarengo.

Boa leitura e sucesso com o Delphi XE! Aproveito para desejar a todos os nossos leitores um ótimo ano de 2011.

**Guinther Pauli**

*guinther@devmedia.com.br*

Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix

Microsoft Certified: MCP,MCAD,MCSD.NET,MCTS,MCPD (ASP.NET,C#)

guintherpauli.blogspot.com - twitter.com/guintherpauli

**Fale com o Editor!**

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

**Guinther Pauli - Editor da Revista**
*guinther@devmedia.com.br*

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
*http://www.devmedia.com.br/clubedelphi/portal.asp*

DevMedia

# Quick Update:

# MonoDevelop e Prism

## Criando aplicações para iPhone com Delphi

**Sebastian P.R. Gingter**
*sebastiang@remobjects.com*
*RemObjects Software*
*É desenvolvedor evangelista da RemObjects Software, instrutor internacional, consultor em desenvolvimento .NET e Delphi. Combinando produtos da RemObjects com seu entusiasmo por novas tecnologias, ajuda desenvolvedores a fazerem softwares de forma mais eficiente, mais divertida e minimizando as barreiras entre diferentes plataformas.*

MonoDevelop é um IDE para .NET gratuito e open-source e possui em suas raízes o também livre e open-source IDE para .NET SharpDevelop. Com SharpDevelop ainda sendo um IDE apenas para Windows utilizando Windows Forms, o MonoDevelop (ou apenas MD, como nós o chamamos) foi portado para #Gtk e massivamente estendido no que diz respeito a compatibilidade multiplataforma. Agora é possível trabalhar sem problemas no Linux, Mac OS X e, é claro, Windows. Você pode usar soluções no Visual Studio com MD e vice-versa, então você terá uma única solução para diferentes plataformas.

Com a integração do Delphi Prism no MonoDevelop nós visamos principalmente o Mac e depois o Windows. Para Windows nós temos a integração do Visual Studio que é, de fato e honestamente, um IDE muito mais confortável e poderoso. Dito isto: é claro que nós queremos estar certos que o MonoDevelop no Windows (MD/Win) também funciona bem, mas MD/Mac é de uma prioridade maior porque você não tem um IDE alternativo.

A combinação MD/Mac vem em forma de uma aplicação Mac completa. Parece que este é o MonoDevelop original mas adicionalmente contém nosso plug-in do Delphi Prism. Isto significa que você terá que instalar os pré-requisitos do MonoDevelop por conta própria (versão do Mono maior ou igual a 2.4).

Depois de iniciar, o Delphi Prism irá lhe informar que não está licenciado e pedirá para registrar sua licença (se você não tem uma licença do Delphi Prism você pode obter sua cópia e uma chave trial de 30 dias no site da Embarcadero).

Quando selecionar "Start a new Solution" você possui algumas opções: a primeira trata-se de uma aplicação console, mas você também pode optar por trabalhar com projetos ASP.NET,

criar aplicações para iPhone ou iPod (certifique-se que você tem o iPhone SDK e o Trial do MonoTouch instalados) e pode-se construir aplicações Moonlight (a alternativa Mono para Silverlight).

Neste artigo nós faremos um aplicativo para iPhone (**Figura 1**). Não porque queremos que você compre e siga o iPhone Developer Program e juntamente compre o MonoTouch da Novell, mas apenas porque isto é uma coisa legal de se mostrar.

Neste diálogo você vê as opções que configurei para o aplicativo "Hello world" no iPhone. Esteja certo de atender todos os requisitos (MD, iPhone SDK e MonoTouch) se você quer seguir o código deste artigo. A próxima coisa que você verá é o MD editor com o projeto aberto e pronto para iniciar.

Não vamos adentrar também na abordagem do MVC do desenvolvimento iPhone, que é também a chave para o desenvolvimento com Cocoa (API do Mac OS X), mas você precisa saber que a UI é separada do seu código. Ao contrário do Delphi ou Windows Forms onde os controles colocados no formulário são automaticamente propriedades da sua classe Form, você precisa definir quais controles e eventos (chamados de ações) estão visíveis para seu código usando as classes definidas.

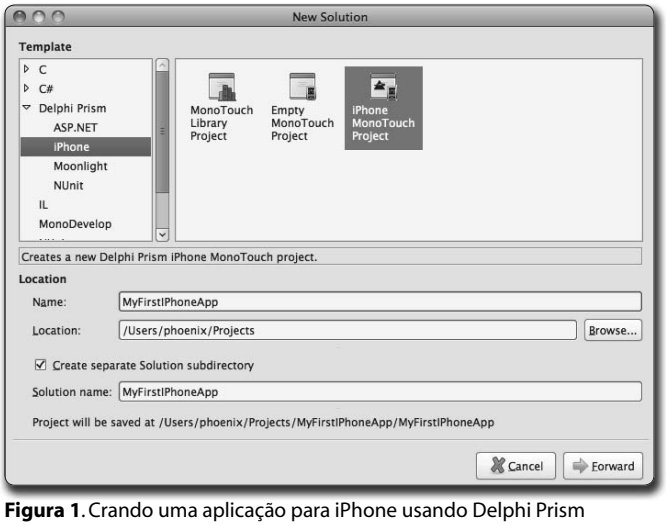



**Figura 1**. Crando uma aplicação para iPhone usando Delphi Prism

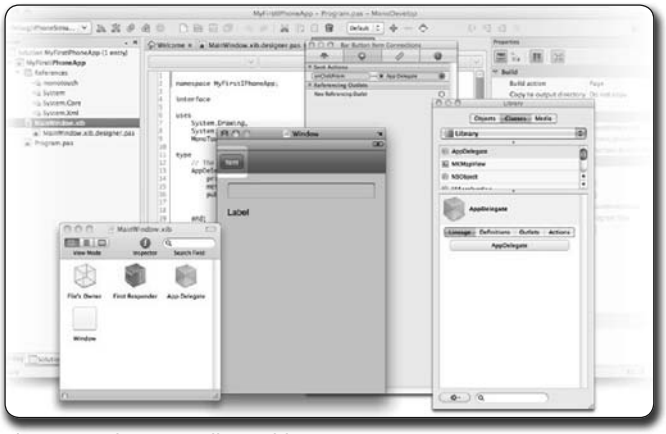


**Figura 2.** Aplicação "Hello World"

A primeira coisa que nós criaremos é a GUI. Nós simplesmente queremos exibir o texto digitado em um TextBox em um Label quando o usuário pressionar um botão, tipicamente um "Hello World" (**Figura 2)**. Na "Solução" na parte esquerda do seu IDE do MonoDevelop nós expandimos o projeto e damos um duplo-clique no arquivo "MainWindow.xib". Este deve iniciar um programa chamado "Interface Builder" que pertence ao iPhone SDK. O arquivo .xib é um pouco comparável aos arquivos .dfm que conhecemos do Delphi ou como arquivos .Designer.pas que definem o layout de um form Windows Forms.

A parte seguinte do design da GUI (conectar saídas e sinais) é de fato mais difícil se você nunca trabalhou com o "interface builder". Na janela Library você seleciona os Objects na parte superior e usa a caixa de busca no rodapé para pesquisar pela Toolbar. Pegue-a e a solte na janela e ela irá se ajustar no topo. Então faça o mesmo para o Text Field e o Label.

Agora que todos os controles estão no lugar é hora de torná-los acessíveis. Na janela Library selecione Classes no topo e digite "App" na busca. Então você pode selecionar a classe AppDelegate que é a interface entre o código da nossa aplicação e a GUI. Ainda na janela Library selecione "Outlets" na metade inferior. Nós vamos definir assessores para o Text Field e o Label agora clicando no "+" abaixo dos pontos listados, entre com o texto e repita para o label. Agora nós vamos tornar nosso evento de click acessível selecionando Actions e adicionando a ação "onClick:" para a classe AppDelegate.

A próxima tarefa é conectar nossas saídas e ações aos controles. Para fazer isto, simplesmente selecione o botão na toolbar (rotulado de "Item"). Na janela de propriedade selecione as conexões (seta branca no círculo azul). Há uma área chamada "Sent actions" com uma entrada para "Selector". Use o mouse, clique no círculo da direita e arraste na caixa amarela "App Delegate" na janela "MainWindow.xib". Soltando você poderá selecionar a ação que você quer para o link com a ação do item "selector" enviada. Para ambos o Text Field e o Label arraste o círculo "new referencing outlet" na aplicação delegando e ligando-os aos pontos e ao label que definimos anteriormente. Agora você pode pressionar Apple +S para salvar o arquivo MainWindow.xib.

O MonoDevelop gera o code-behind para nós automaticamente. Abra o arquivo "MainWindow.xib.designer.pas" (expanda o nó "MainWindow.xib" na sua solução para encontrá-lo) para ver o código gerado. Localize a linha a seguir e copie tudo (sem o empty):

```
method onClick(sender: MonoTouch.UIKit.UIBarButtonItem); partial;
empty;
```

Cole na sessão private da classe AppDelegate no arquivo Program.pas. Este é o método que é chamado quando o evento é acionado na GUI. Na parte de implementação complete o método parcial e preencha em seu corpo:

**Figura 3**. Testando a aplicação Delphi para iPhone

```
method AppDelegate.onClick(sender: UIBarButtonItem);
begin
  label.Text := textbox.Text;
end;
```

De fato isto é tudo. Compile o projeto usando a opção na barra e ferramentas do MD (ou pressionando Apple-B). Seu código será compilado e os assemblys então serão automaticamente compilados em código nativo para iPhone pelo compilador do MonoTouch. Quando clicar em Run ou Debug o simulador do aplicativo iPhone deve aparecer e executar nossa aplicação Hello World (**Figura 3)**.

Como você pode ver construir uma aplicação para iPhone usando Delphi Prism não é apenas trabalhoso, mas você pode fazer uso completo da API do Cocoa que o iPhone SDK lhe oferece. É claro que isto funciona apenas se você possui o iPhone SDK e o MonoTouch, mas para desenvolver para o simulador como fizemos, a versão gratuita de ambos é suficiente. Você necessitaria apenas pagar pela associação de desenvolvedores de programas da Apple e pela licença do MonoTouch se você vai implantar seus programas em um dispositivo físico. Em outras palavras: a menos que você queira iniciar vendendo seu aplicativo e obtendo lucro, você pode trabalhar gratuitamente.

**Dê seu feedback sobre esta edição!**

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

***www.devmedia.com.br/clubedelphi/feedback***

Cursos Online

Assinatura .net magazine PLUS

Mais conteúdo .NET por muito menos!

A Revista **.net Magazine** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade .

**Conheça abaixo os cursos já disponíveis:**
www.devmedia.com.br/curso/netmagazine

- **Crie uma loja Virtual completa**
- **Construindo relatórios com Crystal Reports e Visual Studio 2005**
- **Criando uma aplicação Web Completa**
- **Criando uma aplicação client/server no Visual Studio 2005**
- **Aprenda a criar um blog com ASP.NET**
- **Curso de C#**

**A sua melhor opção de aprendizagem!**

Assine a **.net Magazine** e
Comece já seu treinamento!
www.devmedia.com.br/assine

DevMedia Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038

# Firebird 2.5

## Conheça os principais recursos do banco de dad

O desenvolvimento do SGBD Firebird foi iniciado em Julho do ano 2000 a partir do código fonte da versão beta do InterBase 6.0, disponibilizado pela Inprise (Borland) sob licença Open Source. Desde então cinco versões foram lançadas pela sua equipe de desenvolvimento, mostrando uma evolução contínua do produto. O foco deste artigo é mostrar as principais funcionalidades para desenvolvedores adicionadas à versão 2.5, lançada em outubro de 2010, no ano em que o projeto faz seu décimo aniversário. Antes de aprofundar nestes temas, gostaria de mostrar um breve histórico da evolução do Firebird nestes 10 anos.

Na versão 1.0 a maior parte do trabalho foi interna, já que o código inicial não tinha documentação e não chegava nem mesmo a compilar. Uma das alterações mais importantes nesta versão foi a correção de um bug de segurança do InterBase, que foi rapidamente identificado quando seu código tornou-se público.

Na versão 1.5 o código-fonte foi convertido de C para C++. Algumas novidades que apareceram nesta versão foram variáveis de contexto como CURRENT_USER e CURRENT_ROLE, funções como COALESCE e NULLIF, expressões CASE, o tipo de dados BIGINT, melhorias em alguns comandos DDL (CREATE OR ALTER, RECREATE), o comando EXECUTE STATEMENT, as cláusulas FIRST e SKIP e suporte a savepoints e locks pessimistas.

A versão 2.0 introduziu o conceito de tabelas derivadas, cursores explícitos, valores default para parâmetros, as funções IIF, CHAR_LENGTH, TRIM, RDB$GET_CONTEXT e RDB$SET_CONTEXT, o operador IS [NOT] DISTINCT, os comandos EXECUTE BLOCK e COMMENT ON, a cláusula RETURNING, índices definidos por expressões, Collates (PT_BR e WIN_PTBR) para o Brasil, suporte a Unicode, melhorias de performance relacionadas à nova implementação de índices e diversas mudanças no otimizador, backups físicos e incrementais com o NBackup e suporte a plataformas 64-bits.

A versão 2.1 introduziu dezenas de novas funções internas, Triggers de eventos de conexão e transação, tabelas temporárias, Collates customizáveis com CREATE COLLATION, queries recursivas com Common Table Expression (CTE), os comandos UPDATE OR INSERT e MERGE, a função agregada LIST, a possibilidade de uso de Domains em PSQL, a interoperabilidade entre BLOBs e Strings, tabelas de monitoração, o utilitário fbsvcmgr e diversas melhorias implementadas no protocolo de comunicação, melhorando a performance de comunicação cliente/servidor via internet.

Diante de toda essa evolução, é importante ressaltar que toda alteração e novas funcionalidades do Firebird são pensadas, levando em consideração dois princípios estabelecidos pelo projeto: compatibilidade e padronização SQL. A compatibilidade é sempre levada extremamente a sério, e dificilmente algo que funciona em uma versão deixa de funcionar em outra, respeitando o investimento feito pelos desenvolvedores. Em relação ao padrão SQL, ele é sempre consultado e analisado em detalhes, e quando possível, as implementações são realizadas com base neste padrão.

A versão 2.5, assim como as anteriores, possui diversas novas funcionalidades. Primeiramente, veremos sobre possíveis problemas durante o processo de migração. Em seguida, veremos sobre a nova arquitetura SuperClassic, que permite melhor aproveitamento de recursos e

### Resumo DevMan

***De que trata o artigo:***

O artigo mostra as novidades da nova versão 2.5 do SGBD Firebird. Inicialmente é apresentado um pequeno histórico de sua evolução e logo em seguida são explicadas as novidades com pequenos exemplos de uso que farão com que o leitor os entenda facilmente.

***Para que serve:***

Demonstrar ao leitor através de exemplos como realizar o upgrade de sua versão atual para o Firebird 2.5, bem como ilustrar todas as suas novidades, fazendo com que o leitor compreenda o funcionamento de cada recurso e possa aplicar em seus sistemas.

***Em que situação o tema é útil:***

Tendo conhecimento dos principais recursos suportados pelo sistema gerenciador de banco de dados, o desenvolvedor pode realizar seu trabalho de forma mais eficaz, reduzindo a necessidade de escrever código desnecessário nas aplicações.

***Firebird 2.5:***

Veremos neste artigo um pouco sobre o Firebird 2.5, que possui uma nova arquitetura que possibilita uma melhoria de desempenho em aplicativos de banco de dados. Após a leitura, o leitor se sentirá tentado a migrar suas aplicações para essa versão utilizando todas as diversas novidades incluídas.

# os mais utilizado pela comunidade Delphi

Adriano dos Santos Fernandes

melhor desempenho em máquinas multiprocessadas, e as alterações diretamente relacionadas ao desenvolvimento. Os exemplos presentes nas listagens foram feitos para o ISQL, mas devem funcionar sem problemas em qualquer ferramenta.

### Migrando de versões anteriores

Conforme citado anteriormente, a compatibilidade é levada a sério no Firebird. Infelizmente, às vezes é preciso fazer mudanças que causam certas dificuldades de migração. Antes da versão 2.1 o Firebird tinha importantes bugs referentes a implementação de character sets multibyte herdados do InterBase. Um destes character sets é o UNICODE_FSS, que é usado pelas tabelas de sistema RDB$. O Firebird não convertia, do character set cliente para o UNICODE_FSS, nomes, Strings e código-fonte de objetos durante a execução de comandos DDL e simplesmente gravava os dados, que do ponto de vista do UNICODE_FSS eram inválidos. A partir da versão 2.1 esta conversão passou a ser feita, embora ainda fosse possível a criação de objetos inválidos usando o character set NONE. Para corrigir os objetos gravados de forma incorreta, foi disponibilizado um script que deveria ser rodado após o restore na versão 2.1.

A partir da versão 2.5 o Firebird passou a recusar dados e metadados inválidos com o character set UNICODE_FSS. Isto significaria que qualquer banco de dados de versões anteriores que tivesse problemas de character set não poderia ser restaurado na nova versão. Diante dessa situação, foram acrescentadas duas opções ao gbak, para serem usadas durante o processo de restauração e corrigir este tipo de problema.

A opção -FIX_FSS_METADATA seguida do nome de um character set é usada para informar em que character set foram criados os objetos de banco de dados presentes no backup. Durante a restauração, os metadados são convertidos deste character set para o UNICODE_FSS e gravados corretamente.

A outra opção, -FIX_FSS_DATA seguida do nome de um character set, deve ser usada apenas em bancos de dados que tenham tabelas de usuários criadas usando o character set UNICODE_FSS. Esta opção realiza a conversão dos dados da mesma forma que -FIX_FSS_METADATA realiza a conversão dos metadados.

O uso destas opções só é necessário caso o banco possua metadados e dados com caracteres fora do padrão ASCII, como caracteres acentuados, por exemplo. É recomendado que se tente inicialmente fazer a restauração sem usar estas opções. Se o processo falhar por estes problemas, será mostrada uma mensagem de erro recomendando o uso. Para usar o character set correto nestas opções, o usuário precisa ter um conhecimento mínimo sobre character sets (ou páginas de códigos de caracteres) e dos sistemas operacionais usados pelas aplicações cliente. Por exemplo, aplicações que executavam comandos DDL pelo Windows (configurado para Português/Brasil) devem usar WIN1252. Já no Linux a maioria das distribuições atuais trabalha com UTF-8, que é compatível com UNICODE_FSS e não deveria causar problemas.

Estas opções podem não resolver os problemas caso o banco de dados tenha dados ou metadados gravados de forma incorreta usando diferentes character sets. Neste caso será necessário corrigir os problemas com o auxílio dos scripts de migração usando a versão 2.1.

Após a migração do banco de dados, o desenvolvedor precisa se certificar que seus aplicativos conectam-se ao banco usando o client character set correto. O uso do character set NONE (padrão) ou de um character set incorreto pode causar erros de "malformed string" e "transliteration exception".

**Nota do DevMan**

Common Table Expression (CTE) é similar ao conceito de tabelas derivadas, porém, o nome da tabela é definido no começo da instrução com a cláusula WITH. Exemplo: WITH X AS (SELECT * FROM PESSOAS) SELECT * FROM X. Com as CTEs é possível fazer queries hierárquicas (recursivas) usando-se WITH RECURSIVE e UNION ALL. Veja mais exemplos no release notes da versão 2.1 na sessão Links.

**Nota do DevMan**

Tabela derivada (derived table) é o nome dado pelo padrão SQL a uma query usada na cláusula FROM de outra query. Por exemplo: SELECT X.* FROM (SELECT * FROM PESSOAS) X. Neste caso, X é uma tabela derivada.

**Nota do DevMan**

Comandos DDL (Data Definition Language) são comandos SQL usados para a definição dos objetos do banco de dados. Exemplos de comandos DDL: CREATE TABLE, DROP TABLE e CREATE VIEW.

## A arquitetura SuperClassic

Desde sua primeira versão, o Firebird foi disponibilizado em dois modos de operação (chamados de arquiteturas) diferentes, a SuperServer e a Classic. Na SuperServer as conexões aos bancos de dados são gerenciadas com múltiplas threads em um único processo do SO, enquanto que na Classic é aberto um processo para cada conexão realizada.

A vantagem da arquitetura SuperServer é que diversas conexões a um mesmo banco de dados são gerenciadas com apenas um cache de dados e metadados. Sua desvantagem é que, internamente, a execução de queries é feita de maneira cooperativa, isto é, em determinados pontos uma query paralisa sua própria execução e passa o controle para que outra query continue executando, não aproveitando adequadamente os recursos de máquinas com mais de um processador ou núcleo. Nota: O time de desenvolvedores do Firebird trabalha para remover completamente esta limitação na versão 3, e a SuperClassic é o primeiro passo em busca desse objetivo. Outra desvantagem da SuperServer é que apenas um processo pode abrir cada banco de dados. Isto causava uma limitação em aplicativos que usavam a biblioteca Embedded, como veremos mais adiante.

**Nota**

O compartilhamento de um banco de dados em vários processos Classic (ou Embedded) é permitido apenas quando todos os processos sejam da mesma arquitetura, 32 ou 64 bits. Com a mudança de arquitetura da versão Embedded, o desenvolvedor precisa verificar se o desempenho de seu aplicativo foi influenciado negativamente, pois a configuração padrão de buffers dos bancos de dados são diferentes nas duas arquiteturas. Esta configuração pode ser ajustada alterando-se o valor do parâmetro DefaultDbCachePages no arquivo firebird.conf ou com o comando gfix -buffers em cada banco de dados.

**Nota do DevMan**

No Firebird as colunas COMPUTED BY funcionam como colunas somente leitura e retornam o valor de uma expressão. Um exemplo de coluna COMPUTED BY é uma coluna IDADE que retorne a idade de uma pessoa baseando-se no valor da coluna DATA_NASCIMENTO e a data atual.

Na arquitetura Classic cada processo pode executar queries paralelamente, de forma preemptiva, porém, cada um possui seu exclusivo cache. Para manter a integridade, o Firebird precisa realizar comunicação inter-processo através de seu lock manager. Essa comunicação causa uma perda de desempenho em relação à SuperServer. Nesta arquitetura também existia a possibilidade de trabalhar com diversas conexões em apenas um processo, porém com a mesma limitação presente na SuperServer, onde as queries de um mesmo processo rodavam de maneira cooperativa. Além disso, esta possibilidade existia apenas com a versão Embedded para Linux, pois rodando no modo servidor a criação de novos processos era sempre feita automaticamente a cada conexão.

O novo modo SuperClassic é uma mistura das duas arquiteturas preexistentes. Um processo SuperClassic pode gerenciar diversas conexões usando threads, como no SuperServer. Mas, diferentemente do SuperServer, cada conexão possui seu cache exclusivo, como no Classic. Como o cache não é compartilhado, as queries podem rodar de maneira preemptiva, melhorando o desempenho em máquinas multiprocessadas.

A arquitetura SuperServer também teve uma melhoria. Até a versão 2.1 as queries eram sincronizadas por servidor, e na versão 2.5 são sincronizadas por banco de dados. Isto significa que um servidor 2.5 gerenciando conexões a vários bancos de dados aproveitará os recursos de hardware de forma mais satisfatória.

Além de SuperServer e Classic, o Firebird também é disponibilizado como uma biblioteca: a versão Embedded. Até a versão 2.1 a biblioteca Embedded trabalhava no modo SuperServer no ambiente Windows e no modo Classic em outros SOs. Na versão 2.5 a biblioteca Embedded usa o modo SuperClassic em todos os SOs. Como o SuperClassic é o Classic melhorado, tornou-se possível o acesso simultâneo a um banco de dados por diversos aplicativos usando a biblioteca Embedded, removendo esta limitação que existia em ambientes Windows.

## Character Sets e Collates

No Firebird, diferentes tabelas ou até mesmo diferentes colunas de uma mesma tabela podem usar diferentes character sets e collates. Isto permite uma grande flexibilidade, mas pode causar problemas caso o desenvolvedor ou DBA não se atente às necessidades da aplicação. Por isso, um banco de dados pode ter um character set default, que é automaticamente usado quando não é especificado um character set durante a definição de uma coluna de tipo String. Algo semelhante não existia para Collates, sendo que o Collate default de cada character set é o modo de comparação binária dos bytes que compõem uma String.

Na nova versão foi introduzido o comando ALTER CHARACTER SET, permitindo a alteração do Collate padrão de um character set. Além do comando ALTER CHARACTER SET, agora o Collate padrão do character set padrão do banco pode ser definido no momento da criação do banco de dados. A **Listagem 1** mostra a criação de um banco de dados usando o character set padrão WIN1252 e definindo seu Collate padrão para WIN_PTBR. Em seguida o Collate padrão do character set UTF8 é alterado para UNICODE_CI_AI.

A **Listagem 2** mostra o banco operando com o character set default WIN1252 e o collate WIN_PTBR (case-insensitive).

Outra necessidade comumente encontrada em aplicações é a gravação de códigos alfanuméricos. Até o Firebird 2.1 a ordenação de campos do tipo String era sempre feita no modo de comparação de texto. Isto quer dizer que um código "A10" é listado antes de "A2" caso ordenado por esta coluna. A **Listagem 3** mostra um exemplo desta situação.

O Firebird 2.5 permite que Collates sejam configuráveis. Uma das opções do Collate UNICODE permite que números sejam ordenados por ordem numérica. A **Listagem 4** mostra a criação do Collate UNICODE_NUM com o uso da opção NUMERIC-SORT e a ordenação usando este mesmo Collate.

Outra novidade da versão 2.5 é o Collate UNICODE_CI_AI (para o character set UTF8), variação do Collate UNICODE pré-

configurado para desconsiderar diferenças de acentos e maiúsculas/minúsculas. O Collate UNICODE_CI_AI funciona de maneira similar ao WIN_PTBR e PT_BR, mas aceita o conjunto completo de caracteres Unicode.

## Comandos DDL (Data Definition Language)

Além do comando CREATE COLLATION, mostrado na seção anterior, o Firebird 2.5 acrescenta comandos DDL que aliviam limitações anteriores e outros comandos totalmente novos. Duas limitações existentes nas versões anteriores eram relacionadas à impossibilidade de alteração de colunas COMPUTED BY e Views. Era preciso eliminar estes objetos e recriá-los com as alterações. O problema é que o Firebird não permite que um objeto seja eliminado quando este está sendo usado por outro objeto. Algumas ferramentas (como o Flamerobin, por exemplo) geram scripts para eliminar e recriar objetos durante a alteração de um objeto que não era permitida, porém esta não é a solução ideal. A **Listagem 5** mostra o comando usado para alteração de uma expressão COMPUTED no Firebird 2.5.

A **Listagem 6** mostra um exemplo do comando ALTER VIEW. Assim como em outros comandos DDL, também é permitido o uso dos comandos RECREATE VIEW e CREATE OR ALTER VIEW. Além dos novos comandos relacionados à Views, agora é permitido o uso de Stored procedures na cláusula FROM de uma View, o que não era permitido nas versões anteriores. A **Listagem 7** mostra um exemplo.

Na nova versão também foram incluídos comandos DDL para manuseio do banco de dados de usuários. Estes comandos são: CREATE USER, ALTER USER e DROP USER. Estes comandos podem ser executados quando conectado a qualquer banco de dados, mas sempre atualizam os dados do banco de dados geral de usuários (security2.fdb). Os comandos CREATE USER e DROP USER podem ser usados apenas por usuários com privilégio de administrador. O comando ALTER USER pode ser usado por qualquer usuário, desde que usado para alterar apenas suas próprias infor-

**Listagem 1**. Exemplos de alteração de collate padrão

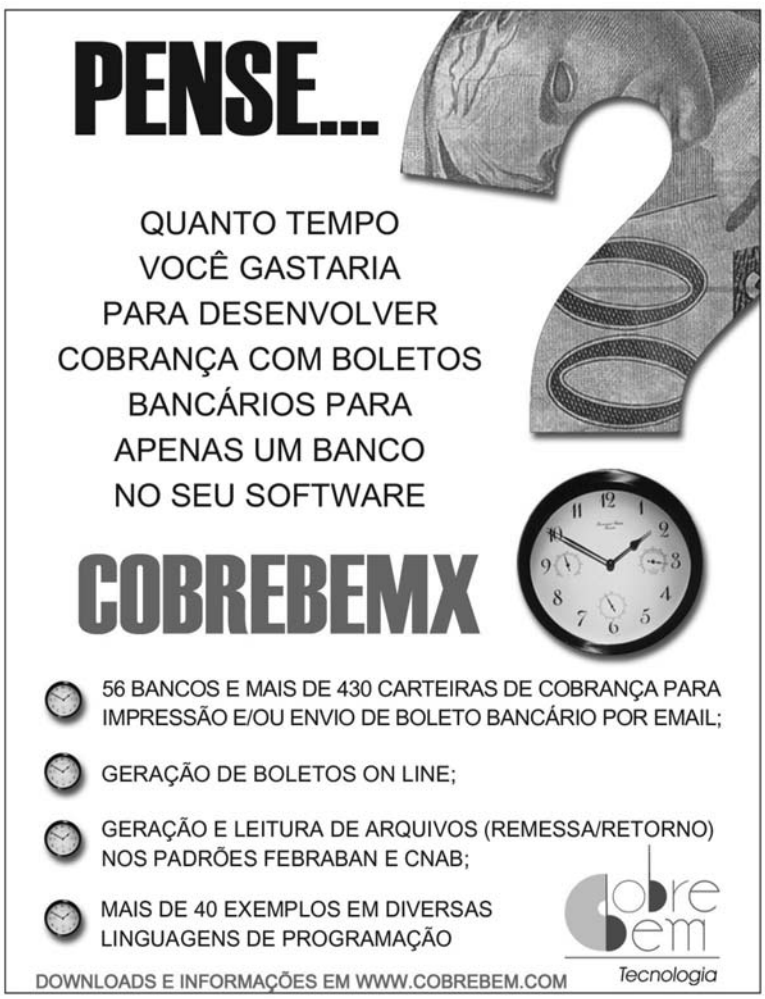
```
CREATE DATABASE 'TEST.FDB'
  DEFAULT CHARACTER SET WIN1252
  COLLATION WIN_PTBR;
ALTER CHARACTER SET UTF8
  SET DEFAULT COLLATION UNICODE_CI_AI;
```

**Listagem 2**. Comparação de strings usando o collate WIN_PTBR

```
CREATE TABLE PESSOAS (NOME VARCHAR(20));
INSERT INTO PESSOAS VALUES ('Fulano');
INSERT INTO PESSOAS VALUES ('Beltrano');
-- Localiza Fulano (= FULANO)
SELECT * FROM PESSOAS WHERE NOME = 'FULANO';
```

**Listagem 3**. Ordenação de códigos alfanuméricos usando o collate UNICODE

```
CREATE TABLE DOCUMENTOS (
  CODIGO VARCHAR(10) CHARACTER SET UTF8 COLLATE UNICODE);
INSERT INTO DOCUMENTOS VALUES ('A1');
INSERT INTO DOCUMENTOS VALUES ('A2');
INSERT INTO DOCUMENTOS VALUES ('A10');
INSERT INTO DOCUMENTOS VALUES ('A11');
INSERT INTO DOCUMENTOS VALUES ('A100');
INSERT INTO DOCUMENTOS VALUES ('B1');
INSERT INTO DOCUMENTOS VALUES ('B10');
SELECT CODIGO FROM DOCUMENTOS ORDER BY CODIGO;
-- Resultado: A1, A10, A100, A11, A2, B1, B10
```



mações (a senha, por exemplo). Já os administradores podem usar ALTER USER para alterar informações de qualquer usuário. Além da senha (PASSWORD), é possível registrar os nomes (FIRSTNAME, MIDDLENAME e LASTNAME) e definir se um usuário é ou não um administrador com a cláusula GRANT/REVOKE ADMIN ROLE. A **Listagem 8** mostra exemplos de criação, alteração e remoção de usuários.

Para um usuário tornar-se efetivamente um administrador, um outro administrador (como o SYSDBA) precisa, além de colocar GRANT ADMIN ROLE no CREATE USER ou ALTER USER, conceder a role RDB$ADMIN com o comando "GRANT RDB$ADMIN TO [user]" em cada banco de dados. O usuário também precisará conectar-se ao banco de dados usando a ROLE RDB$ADMIN. A **Listagem 9** mostra o ADMIN_ADJUNTO atuando como SYSDBA.

**Nota**

A versão 2.1.4 também deve ser liberada com esta novidade.

## Expressões, literais e funções

Na parte de expressões SQL, o Firebird 2.5 acrescenta melhorias e novidades. A função agregada LIST (presente desde a versão 2.1) agora aceita qualquer expressão em seu segundo parâmetro, onde o desenvolvedor pode especificar a String de separação dos elementos retornados. Até a versão 2.1.3 era possível apenas o uso de Strings constantes nesse parâmetro. A **Listagem 10** mostra a utilidade deste parâmetro, principalmente no desenvolvimento de relatórios, agrupando várias mensagens referentes a um documento em um mesmo registro e mostrando cada mensagem em uma linha.

Referente à parâmetros de comandos SQL, o Firebird suporta apenas o uso de parâmetros anônimos com o símbolo "?" (ponto de interrogação). Algumas bibliotecas (inclusive de Delphi) aceitam o uso de parâmetros nomeados usando ":" (dois pontos). Quando o desenvolvedor tentava escrever queries usando o padrão [WHERE :CODIGO IS NULL OR CODIGO = :CODIGO] para trazer todos os registros quando o parâmetro não fosse informado, o Firebird retornava um erro. Isto porque este comando é traduzido pelas bibliotecas para [WHERE ? IS NULL OR CODIGO = ?] e o Firebird não aceitava o uso de parâmetro com o predicado IS NULL, pois o tipo do parâmetro era considerado como desconhecido. Na versão 2.5 foi adicionado à API o tipo SQL_NULL. As bibliotecas de acesso precisam entender este novo tipo e apenas passar se o valor do parâmetro é ou não NULL, permitindo assim o uso deste padrão de comando.

Outra novidade referente a expressões é o novo predicado SIMILAR TO. Esta expressão é usada para fazer comparações usando expressões regulares de acordo com o padrão SQL. A **Listagem 11** mostra uma verificação de números de telefones cadastrados fora do padrão (NN) NNNN-NNNN. O caractere de escape funciona exatamente como no comando LIKE, considerando o próximo caractere como um valor literal, ao invés de usá-lo como um operador.

A **Tabela 1** mostra os operadores permitidos em expressões regulares e a **Tabela 2** mostra as classes de caracteres que podem ser usadas com o operador [[:CLASSE:]].

O Firebird 2.5 também suporta novos formatos de valores literais (constantes) para números e strings. Agora é possível escrever números inteiros no formato hexadecimal, usando o prefixo 0x. Quando o número possuir até 8 dígitos hexadecimais após o 0x, como em 0xFFFFFFFF, o núme-

**Listagem 4**. Ordenação usando a opção NUMERIC-SORT do collate UNICODE

```
CREATE COLLATION UNICODE_NUM
  FOR UTF8 FROM UNICODE 'NUMERIC-SORT=1';
CREATE TABLE DOCUMENTOS2 (
  CODIGO VARCHAR(10) CHARACTER SET UTF8 COLLATE UNICODE_NUM);
INSERT INTO DOCUMENTOS2 SELECT * FROM DOCUMENTOS;
SELECT CODIGO FROM DOCUMENTOS2 ORDER BY CODIGO;
-- Resultado: A1, A2, A10, A11, A100, B1, B10
```

**Listagem 5**. Exemplo de alteração de expressão COMPUTED BY

```
-- Criação da tabela com erro na expressão da coluna IDADE.
CREATE TABLE PESSOAS (
  NOME VARCHAR(60),
  DATA_NASCIMENTO DATE,
  IDADE COMPUTED BY (DATEDIFF(YEAR, CURRENT_DATE, DATA_NASCIMENTO)));
INSERT INTO PESSOAS VALUES ('Fulano', '2000-01-01');
INSERT INTO PESSOAS VALUES ('Beltrano', '1950-05-10');
SELECT * FROM PESSOAS;  -- Lista a idade como negativa.
-- Correção da expressão da coluna IDADE.
ALTER TABLE PESSOAS
  ALTER IDADE COMPUTED BY (DATEDIFF(YEAR, DATA_NASCIMENTO, CURRENT_DATE));
SELECT * FROM PESSOAS;
```

**Listagem 6**. Exemplo do comando ALTER VIEW

```
-- View criada incorretamente sem a expressão WHERE.
CREATE VIEW PESSOAS_MAIORES AS
  SELECT * FROM PESSOAS;
-- Correção da view.
ALTER VIEW PESSOAS_MAIORES AS
  SELECT * FROM PESSOAS WHERE IDADE >= 18;
```

**Listagem 7**. Exemplo de uso de stored procedure em uma view

```
SET TERM !;
CREATE PROCEDURE PESSOAS_SP RETURNS (NOME VARCHAR(60), IDADE INTEGER) AS
BEGIN
  FOR SELECT NOME, IDADE FROM PESSOAS INTO NOME, IDADE DO
     SUSPEND;
END!
SET TERM ;!
CREATE VIEW PESSOAS_SP_MAIORES AS
  SELECT * FROM PESSOAS_SP WHERE IDADE >= 18;
```

## Nota do DevMan

Complemento de dois é a notação mais comum usada para representar números inteiros com sinal em sistemas computacionais. Nesta notação o bit mais significativo de um número positivo ou do número zero é representado como 0. Os números negativos são representados com os bits invertidos e somado o valor 1. Desta forma, em um número de 32 bits o 1 é representado com trinta e um bits 0 seguido por um bit 1, enquanto que o número -1 é representado por trinta e dois bits 1.

ro adquire o tipo INTEGER (32 bits com sinal). Quando possuir mais de 8 dígitos, como em 0x0FFFFFFFF, o tipo adquirido é o BIGINT (64 bits com sinal). Desta forma, estas duas constantes retornam valores diferentes, pois os números negativos são armazenados na notação complemento de dois.

Além de números, também é possível a criação de Strings binárias (character set OCTETS) usando a notação hexadecimal x''. Em ambos os casos a letra X pode ser escrita em maiúscula ou minúscula. A **Listagem 12** mostra o uso destes novos tipos de valores literais. Note que cada par de caracteres hexadecimais se transforma em um byte na String resultante.

A nova função BIN_NOT se junta ao grupo de funções binárias adicionadas na versão 2.1, BIN_AND e BIN_OR. Junto com as constantes hexadecimais, essa família de funções facilita o uso de máscaras de valores binários gravados em um único campo, técnica comumente utilizada em aplicações e agora facilitada no banco de dados. A **Listagem 13** mostra o uso da função BIN_NOT, que inverte todos os bits de um valor inteiro.

Na versão 2.1 foi adicionada a função GEN_UUID, que retorna um UUID como um valor do tipo CHAR(16) CHARACTER SET OCTETS. Este tipo de retorno foi escolhido, pois é a representação mais compacta para ser usada em índices, porém, requer suporte das aplicações que desejam exibir estes UUIDs aos usuários. Na nova versão foram adicionadas funções para conversão entre as representações binária e texto [CHAR(36) CHARACTER SET ASCII] de UUIDs, as funções CHAR_TO_UUID e UUID_TO_CHAR. A **Listagem 14** mostra o uso destas funções.

**Listagem 8**. Manuseio do banco de dados de segurança com comandos SQL

```
CREATE USER FULANO PASSWORD 'altereja'
  FIRSTNAME 'Fulano' LASTNAME 'da Silva';
ALTER USER SYSDBA PASSWORD 'masterkey';
DROP USER BELTRANO;
CREATE USER ADMIN_ADJUNTO PASSWORD 'masterkey' GRANT ADMIN ROLE;
GRANT RDB$ADMIN TO ADMIN_ADJUNTO;
```

**Listagem 9**. Conectando-se com privilégios de administrador a um banco de dados

```
CONNECT 'TEST.FDB' USER ADMIN_ADJUNTO
  PASSWORD 'masterkey' ROLE RDB$ADMIN;
DROP USER FULANO;
```

**Listagem 10**. Usando a função LIST com expressões no segundo parâmetro

```
SELECT LIST(MENSAGEM, ASCII_CHAR(13) || ASCII_CHAR(10))
  FROM MENSAGENS_DOCUMENTO WHERE DOCUMENTO = 10;
```

**Listagem 11**. Verificando números de telefones cadastrados fora do padrão

```
SELECT * FROM PESSOAS
  WHERE TELEFONE NOT SIMILAR TO
    '\([0-9]{2}\) [0-9]{4}\-[0-9]{4}' escape '\'
```

**Listagem 12**. Novos formatos de expressões literais

```
SELECT 0xDEADBEEF, x'DEADBEEF' FROM RDB$DATABASE;
SELECT OCTET_LENGTH(x'DEADBEEF') FROM RDB$DATABASE;
-- Resultado de OCTET_LENGTH: 4, e não 8
```

| Operador | Descrição |
|---|---|
| X{2} | Duas ocorrências de X. |
| X{2,} | Duas ou mais ocorrências de X. |
| X{4,6} | De quatro a seis ocorrências de X. |
| X? | Zero ou uma ocorrência de X. |
| X* | Zero ou mais ocorrências de X. |
| X+ | Uma ou mais ocorrências de X. |
| X\|Y | X ou Y. |
| _ | Qualquer caractere - como no LIKE. |
| % | Qualquer sequência de caracteres - como no LIKE. |
| (X) | Agrupa X para ser tratado pelo operador subsequente. |
| [XYZ] | Qualquer caractere igual a X, Y ou Z. |
| [^XYZ] | Qualquer caractere diferente de X, Y ou Z. |
| [X-Z] | Qualquer caractere entre X e Z. |
| [[:CLASSE:]] | Qualquer caractere de uma certa classe, conforme Tabela 2. |

**Tabela 1.** Operadores de expressões regulares

| Classe | Descrição |
|---|---|
| ALPHA | Qualquer caractere entre A e Z. |
| UPPER | Qualquer caractere maiúsculo. |
| LOWER | Qualquer caractere minúsculo. |
| DIGIT | Dígitos de 0 a 9. |
| SPACE | Espaço: caractere (ASCII_CHAR) 32. |
| WHITESPACE | Todo tipo de espaço: caracteres 9, 10, 11, 12, 13 e 32. |

**Tabela 2.** Classes de caracteres para expressões regulares

**Listagem 13**. Exemplo de uso da função BIN_NOT

```
SELECT BIN_NOT(0xFFFFFFFF) FROM RDB$DATABASE;  -- Resultado: 0
```

**Listagem 14**. Exemplo de uso das funções UUID_TO_CHAR e CHAR_TO_UUID

```
SELECT UUID_TO_CHAR(UUID), DESCRICAO FROM OBJETOS
 WHERE UUID = CHAR_TO_UUID(?);
```

**Listagem 15**. Usando subqueries como expressões PSQL

```
SET TERM !;
EXECUTE BLOCK RETURNS (COD VARCHAR(10)) AS
BEGIN
  IF ((SELECT COUNT(*) FROM DOCUMENTOS) > 3) THEN
  BEGIN
    COD = (SELECT MAX(CODIGO) FROM DOCUMENTOS);
    SUSPEND;
  END
END!
SET TERM ;!
```

**Listagem 16**. Usando a cláusula TYPE OF COLUMN

```
EXECUTE BLOCK RETURNS (COD TYPE OF COLUMN DOCUMENTOS.CODIGO) AS
BEGIN
  IF ((SELECT COUNT(*) FROM DOCUMENTOS) > 3) THEN
  BEGIN
    COD = (SELECT MAX(CODIGO) FROM DOCUMENTOS);
    SUSPEND;
  END
END!
```

**Nota do DevMan**

UUID (Universally Unique Identifier) é uma sequência aleatória de 16 bytes que, independente do local e momento que seja gerada, é única. Uma das utilidades dos UUIDs é criar chaves em sistemas distribuídos, como filiais de uma empresa com bancos de dados separados e que tenham os dados agregados em um banco principal através de replicação.

**Nota**

Este tipo de segurança não impede que o usuário SYSDBA tenha acesso ao banco, pois os administradores podem desativar os Triggers de banco de dados conectando-se usando a opção -nodbtriggers do ISQL. O problema é que alguns desenvolvedores gostariam de gravar uma entrada em uma tabela de log registrando o evento, mas a exceção causava o ROLLBACK da criação deste registro. Esta e outras situações (como comunicação com outros sistemas através de UDFs) agora podem se beneficiar de um novo comando, o IN AUTONOMOUS TRANSACTION. Assim como um IF ou WHILE, este comando aceita um subcomando (ou vários, com o uso de BEGIN END). No início da execução do IN AUTONOMOUS TRANSACTION é aberta uma nova transação e o subcomando é executado dentro desta transação. Caso o subcomando execute por completo, é feito um COMMIT na transação criada. Caso ocorra uma exceção, é feito um ROLLBACK.

## Linguagem PSQL

A linguagem procedural do Firebird, a PSQL, foi incrementada com novos comandos e outras melhorias na linguagem. A primeira apresentada se trata do uso de subqueries como expressões PSQL. Antes, o desenvolvedor era obrigado a criar uma variável e fazer um SELECT INTO para obter um valor e usá-lo posteriormente em outro comando. Agora as subqueries fazem parte do conjunto de expressões PSQL, e podem ser usadas da mesma maneira que já podiam ser usadas em comandos SQL. A **Listagem 15** mostra o uso de uma subquery em uma comparação e em seguida na atribuição de um valor retornado a uma variável.

Na **Listagem 15** o parâmetro de retorno COD foi definido com o tipo VARCHAR(10) e recebe um valor lido da coluna CODIGO da tabela DOCUMENTOS. Uma alteração do tipo desta coluna poderia deixar este bloco de código inconsistente ou inválido. No Firebird 2.1 foi adicionado o suporte ao uso de Domains em PSQL. Neste caso, o desenvolvedor precisaria criar um domínio e usá-lo na definição da tabela e na definição do parâmetro de retorno do EXECUTE BLOCK para ter um código consistente. Dependendo da situação, a criação de domínios para todas as colunas pode se tornar uma tarefa extremamente burocrática. Pensando nisso, o Firebird 2.5 adiciona a cláusula TYPE OF COLUMN. Com essa cláusula é possível declarar parâmetros e variáveis ou fazer Casts usando o tipo definido de uma coluna. Ao alterar o tipo da coluna, a alteração se reflete automaticamente em todas as Stored Procedures e Triggers que façam uso de seu tipo. A **Listagem 16** mostra o mesmo código da **Listagem 15**, mas usando TYPE OF COLUMN para o parâmetro COD.

No Firebird 2.1 foram acrescentados os Triggers de eventos de banco de dados, como o evento ON CONNECT. Uma das funcionalidades que podiam ser feitas com este tipo de Trigger era a rejeição de uma conexão, permitindo que os desenvolvedores implementassem um sistema de segurança dentro do banco de dados. Para rejeitar uma conexão é necessário lançar uma exceção dentro da Trigger. A **Listagem 17** apresenta uma Trigger ON CONNECT que aceita apenas conexões dos usuários SYSDBA e ADMIN e grava o nome dos outros usuários que tiverem a conexão rejeitada. A exceção usada para rejeitar a conexão é lançada fora do bloco IN AUTONOMOUS TRANSACTION, que já havia terminado e, consequentemente, inserido o registro e finalizado a transação com COMMIT.

O comando EXECUTE STATEMENT ganha algumas novidades importantes na versão 2.5. Uma delas é uma das funcionalidades mais pedidas pelos usuários: ler e atualizar dados em outros bancos de dados. Através da cláusula ON EXTERNAL DATA SOURCE é possível especificar o banco de dados em que o comando executará. É permitido também que o desenvolvedor especifique o nome, senha e role do usuário para fazer a conexão, e o funcionamento da transação iniciada para rodar o comando no banco de dados externo. Com WITH COMMON TRANSACTION (opção default), a transação da

conexão externa é agrupada à transação interna atual e o comando COMMIT ou ROLLBACK executado na transação atual é propagado às transações externas. Com WITH AUTONOMOUS TRANSACTION o funcionamento se torna similar as transações autônomas, sendo feito o COMMIT ou ROLLBACK logo após a execução sucedida ou não do comando externo. A **Listagem 18** mostra o uso deste comando para ler e gravar dados em outro banco de dados.

A **Listagem 18** mostra como era feita a criação de Strings de SQL dinâmico, usando concatenação de Strings. A criação de comandos com concatenação é insegura, pois um usuário mal-intencionado pode gerar comandos não previstos pelo desenvolvedor, causando o ataque conhecido como injeção de SQL.

No novo EXECUTE STATEMENT é possível passar parâmetros de duas formas diferentes. A **Listagem 19** mostra o uso de parâmetros anônimos, com o símbolo de interrogação. Além da passagem de parâmetros anônimos, o comando EXECUTE STATEMENT agora suporta o uso de parâmetros nomeados. As vantagens dos parâmetros nomeados são que o desenvolvedor não precisa se preocupar com a ordem e também pode usar um mesmo parâmetro mais de uma vez sem a necessidade de criação de variáveis.

O uso dos parâmetros dentro da String é similar à passagem de parâmetros feita com o caractere ":" em bibliotecas Delphi, como comentado anteriormente. Os valores são associados aos nomes dos parâmetros usando-se o operador ":=" (dois pontos e igual, assim como no Delphi). A **Listagem 20** mostra o uso de parâmetros nomeados.

**Listagem 17**. Usando o comando IN AUTONOMOUS TRANSACTION

```
CREATE TABLE CONN_LOG (USUARIO VARCHAR(128), DATA TIMESTAMP);
CREATE EXCEPTION E_CONN 'Acesso negado.';
SET TERM !;
CREATE TRIGGER T_CONN ON CONNECT AS
BEGIN
  IF (CURRENT_USER <> 'SYSDBA' AND CURRENT_USER <> 'ADMIN') THEN
  BEGIN
    IN AUTONOMOUS TRANSACTION DO
      INSERT INTO CONN_LOG VALUES (CURRENT_USER, CURRENT_TIMESTAMP);
    EXCEPTION E_CONN;
  END
END!
SET TERM ;!
```

**Listagem 18**. Executando comandos em outros bancos de dados com EXECUTE STATEMENT

```
SET TERM !;
EXECUTE BLOCK RETURNS (NOME VARCHAR(60)) AS
  DECLARE BANCO VARCHAR(60) = 'localhost:/db/pessoas.fdb';
  DECLARE USUARIO VARCHAR(10) = 'SYSDBA';
  DECLARE SENHA VARCHAR(10) = 'masterkey';
  DECLARE NOME_INSERIR VARCHAR(20) = 'Sicrano';
BEGIN
  FOR EXECUTE STATEMENT 'SELECT NOME FROM PESSOAS'
      ON EXTERNAL DATA SOURCE BANCO
      AS USER USUARIO PASSWORD SENHA
      INTO NOME
  DO
  BEGIN
    SUSPEND;
  END

  EXECUTE STATEMENT
    'INSERT INTO PESSOAS (NOME) VALUES (''' || NOME_INSERIR || ''')'
    ON EXTERNAL DATA SOURCE BANCO
    AS USER USUARIO PASSWORD SENHA
    WITH AUTONOMOUS TRANSACTION;
END!
SET TERM ;!
```

**Listagem 19**. Usando EXECUTE STATEMENT com parâmetros anônimos

```
SET TERM !;
EXECUTE BLOCK AS
  DECLARE NOME VARCHAR(20) = 'Sicrano';
  DECLARE DATA_NASCIMENTO DATE = DATE '2000-01-01';
BEGIN
  EXECUTE STATEMENT
    ('INSERT INTO PESSOAS (NOME, DATA_NASCIMENTO) VALUES (?, ?)')
    (NOME, DATA_NASCIMENTO);
END!
SET TERM ;!
```

## API

Referente à API, a principal novidade é a função fb_cancel_operation, que permite que a aplicação cancele a execução de uma query em processamento. O primeiro parâmetro recebe o status vector, seguido pelo handle da conexão e uma opção que especifica a ação a ser tomada. A opção fb_cancel_raise cancela a query atualmente em execução. A query cancelada retorna o código de erro isc_canceled em seu status vector. A opção fb_cancel_abort faz uma quebra forçada da conexão ao banco de dados. As opções fb_cancel_enable e fb_cancel_disable habilitam e desabilitam a possibilidade de cancelamento com fb_cancel_raise, podendo ser usadas para demarcar trechos da aplicação que não devam ser cancelados. Por padrão o cancelamento com fb_cancel_raise é habilitado.

### Nota do DevMan

Injeção de SQL é o nome dado de um ataque em que o usuário mal-intencionado preenche dados de um formulário de forma que um comando gerado no servidor através de concatenação de Strings seja alterado. Geralmente o ataque é feito usando-se aspas simples e um restante de código que termine o comando iniciado pelo desenvolvedor, seguindo com outro comando completo e o símbolo de início de comentário (--) para invalidar o restante do comando.

**Listagem 20**. Usando EXECUTE STATEMENT com parâmetros nomeados

```
SET TERM !;
EXECUTE BLOCK AS
  DECLARE NOME VARCHAR(20) = 'Sicrano';
  DECLARE DATA_NASCIMENTO DATE = DATE '2000-01-01';
BEGIN
  EXECUTE STATEMENT
    ('INSERT INTO PESSOAS (NOME, DATA_NASCIMENTO)
       VALUES (:P_NOME, :P_DATA_NASCIMENTO)')
    (P_DATA_NASCIMENTO := DATA_NASCIMENTO, P_NOME := NOME);
END!
SET TERM ;!
```

## Conclusão

A versão 2.5 acrescenta diversas melhorias ao Firebird, prometendo facilitar o trabalho de DBAs e melhorar a qualidade e o desempenho dos aplicativos feitos pelos desenvolvedores. Sempre melhorando e diante de fatos ocorridos recentemente, como incertezas em relação ao futuro do MySQL, o Firebird ganha cada vez mais espaço no mercado de SGBDs Open Source.

Controlado pelo Projeto Firebird, composto pela união de pessoas escolhidas usando o princípio da meritocracia, o Firebird não pode ser vendido a uma organização comercial que tenha como principal objetivo a obtenção de lucros. Seu desenvolvimento é realizado principalmente por desenvolvedores financiados pela Firebird Foundation, organização que recebe doações de empresas e indivíduos, inclusive brasileiros, e mantém a equipe de desenvolvedores que trabalham para o projeto. Se você ou sua empresa se beneficia financeiramente com o uso do Firebird, o ideal seria que a fundação ou o projeto recebesse em troca uma contribuição financeira ou técnica que pudesse ajudar o Firebird a melhorar cada vez mais.

Além das novidades para desenvolvedores apresentadas neste artigo, a nova versão contém outras melhorias na parte de administração, como novas tabelas de monitoração e a possibilidade de monitorar tudo o que acontece no servidor com o serviço de trace. O leitor pode se aprofundar nestes temas através do release notes da versão.

**Adriano dos Santos Fernandes**
*adrianosf@gmail.com*
*asfernandes.blogspot.com*
*twitter.com/adrianosf*
Adriano dos Santos Fernandes é bacharel em Ciência da Computação pela Universidade Paulista e é um dos desenvolvedores responsáveis pelo Firebird. Atua no projeto desde 2004, sendo desde 2006 contratado pela Firebird Foundation. Trabalha também na empresa Metalúrgica Mococa, desenvolvendo sistemas em Java com banco de dados Oracle.


**Site do Firebird**
*http://www.firebirdsql.org*

**Release notes das últimas versões**
*http://www.firebirdsql.org/rlsnotesh/rlsnotes20.html*
*http://www.firebirdsql.org/rlsnotesh/rlsnotes210.html*
*http://www.firebirdsql.org/rlsnotesh/rlsnotes25.html*

**Páginas da Firebird Foundation**
*http://www.firebirdsql.org/index.php?op=ffoundation&id=main_pt*

**Dê seu feedback sobre esta edição!**

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

***www.devmedia.com.br/clubedelphi/feedback***


**Conhecimento faz diferença!**

**Faça já sua assinatura digital ! | www.devmedia.com.br/es**

**Faça um *upgrade* em sua carreira**

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviçoes); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

# FÓRUM DEVMEDIA

**O lugar perfeito para você ficar por dentro de tudo o que acontece nas tecnologias do mercado atual**

**No novo Fórum da DevMedia você vai encontrar canais específicos de Delphi, ASP.NET, Java, Banco de Dados e Engenharia de Software; além de ter contato com profissionais qualificados da área para troca de informações, sugestões e muito mais.**

**ACESSE AGORA**
**www.devmedia.com.br/forum**

DevMedia

# DataSnap no Delph

## Aplicações distribuídas baseadas em REST e

O conceito de aplicação em camadas é poder ter em uma aplicação, geralmente chamada de servidor de aplicação, todas as suas regras de negócio e permitir que essas regras sejam consumidas por aplicações clientes. O desenvolvimento de aplicações em camadas sempre foi um capítulo à parte no Delphi. No início a tecnologia era chamada de MIDAS logo depois foi batizada de DataSnap. Quando começou, a tecnologia era dependente de COM, porém com o lançamento do Delphi 2009 ela passou a funcionar sobre o framework DBExpress. A partir do lançamento do Delphi 2010 a arquitetura DataSnap deu um novo salto, agora podemos criar servidores DataSnap REST e trafegar dados com notação JSON, tudo isso nativamente no Delphi. Agora com o lançamento do RAD Studio XE uma série de novidades está disponível no novo DataSnap.

REST é a abreviação de Representational State Transfer ou Transferência de estado representacional. Ele surgiu da tese de doutorado sobre a Web escrita por Roy Fielding, que foi um dos idealizadores do protocolo HTTP. Alguns entendem que o REST é um novo protocolo, outros o chamam de arquitetura e alguns defendem que ele é apenas uma filosofia, uma forma de pensar, um estilo de projetar uma aplicação. O REST é uma boa prática onde o principal objetivo é construir aplicações fracamente acopladas, voltadas principalmente para o ambiente Web.

O Delphi 2009 trouxe uma grande mudança na arquitetura do DataSnap. Nesta versão a tecnologia COM foi abolida, mas algo tinha que substituí-la. O que a equipe de desenvolvimento do Delphi fez foi algo extremante simples. O Delphi já possuía uma tecnologia de acesso a dados muito eficiente, totalmente orientada a objetos e 100% escrita em Delphi, o DBExpress. O que eles fizeram foi criar um novo driver para o DBExpress, mas ao invés deste driver conectar a um banco de dados qualquer, ele iria se conectar a um servidor de aplicação. Assim, a partir do Delphi 2009 a conexão do cliente com o servidor de aplicação passou a ser feita utilizando DBExpress trafegando os dados utilizando o protocolo TCP/IP.

A grande maioria dos sistemas em camadas desenvolvidos com Delphi tem o cenário Delphi to Delphi, ou seja, servidor de aplicação Delphi e cliente desenvolvido em Delphi. Mas você pode ser perguntar: então ficamos restritos a clientes Desktop, correto? Errado, para aqueles que desejassem desenvolver clientes Web podem optar pelo Delphi Prism. Para nossa felicidade o Delphi Prism possui um driver DBExpress específico para trabalhar com

### Resumo DevMan

***De que trata o artigo:***

O artigo trata do desenvolvimento de servidores de aplicação utilizando a arquitetura DataSnap XE disponível no RAD Studio XE. Veremos como disponibilizar serviços que possam ser consumidos utilizando o conceito REST.

***Para que serve:***

Aplicações em camadas servem quando queremos concentrar nossas regras de negócio em um único lugar, encapsulando a implementação, facilitando a manutenção e aumentando a escalabilidade. Com isso ficamos livres para criar interfaces, sejam elas Web ou Desktop, para consumir estas regras ou mesmo consumi-las via HTTP com o suporte a REST e JSON e permitimos assim que clientes desenvolvidos em diferentes linguagens possam utilizar nossos serviços.

***Em que situação o tema é útil:***

Aplicações em camadas são úteis, pois este tipo de arquitetura nos permite ter um sistema mais escalável e de fácil manutenção. Ao criar um cliente Web estamos na verdade aumentando consideravelmente o nível de escalabilidade e interoperabilidade, permitindo que nossas regras possam ser consumidas de qualquer lugar sem necessidade de instalação ou outro requisito específico.

***DataSnap no Delphi XE:***

O novo Delphi XE está repleto de novos assistentes e funcionalidades para trabalhar com modelos de arquiteturas modernas, baseados em REST, Javascript e AJAX. Além disso, o Delphi XE gera boa parte do código necessário para a comunicação remota. Este artigo apresenta estes novos recursos, além de um exemplo de como criar uma aplicação servidora com métodos personalizados que podem ser consumidos a partir de aplicações clientes.

### Nota do DevMan

Component Object Model (COM) é uma plataforma da Microsoft para componentes de software lançada em 1993. Ela é usada para permitir a comunicação entre processos e a criação dinâmica de objetos em qualquer linguagem de programação que suporte à tecnologia. O termo COM é frequentemente usado no desenvolvimento de software para se referir a um grupo de tecnologias que incluem OLE, OLE Automation, ActiveX, COM+ e DCOM.

# i XE

# JavaScript

Rodrigo Carreiro Mourão

DataSnap e podemos facilmente consumir um servidor de aplicação feito com Delphi Win32. Uma ótima solução por exemplo é criar interfaces Web em ASP.NET que consomem servidores de aplicação Win32.

É claro que não ficamos limitados apenas a trabalhar com Delphi Prism, com o lançamento do Delphi 2010 foi adicionada a arquitetura DataSnap suporte a transporte de dados sobre HTTP. A partir desta versão é possível desenvolver servidores de aplicação de forma que estes servidores possam ser consumidos por um cliente desenvolvido por praticamente qualquer linguagem de programação, graças a arquitetura REST e aos dados trafegados em notação JSON.

Com o lançamento do RAD Studio XE todos os benefícios da arquitetura DataSnap foram mantidos. Podemos perfeitamente nesta nova versão trafegar dados de qualquer tipo suportado pelo DBExpress, incluindo TDBXReader, bem como qualquer objeto que seja descendente de TJSONObject. É bom enfatizar isso para demonstrar que o compromisso que a Borland tinha em manter compatibilidade com versões anteriores foi mantido pela Embarcadero.

Além da opção DataSnap Server disponível no Delphi 2009 e 2010 agora há a possibilidade de criar servidores baseados na arquitetura REST ou baseados na arquitetura de WebBroker. Além disso, quando criar uma aplicação cliente pode-se lançar mão do Wizard para criação de uma aplicação cliente padrão ou baseada também em REST. Não é foco deste artigo destacar em detalhes a diferença entre os servidores e clientes baseados em REST e os demais, porém basicamente aplicações baseadas em REST têm seu foco nos métodos do protocolo HTTP, a saber: GET, POST, PUT e DELETE.

### Nota do DevMan

Para quem não conhece, o Delphi Prism é a ferramenta para desenvolvimento multiplataforma da Embarcadero, incluindo Windows, Linux e Mac, usando .NET e Mono.

### Nota do DevMan

JSON (Javascript Object Notation) é uma espécie de padronização para transporte de informações entre ambientes heterogêneos e que tem como linguagem o Javascript. Bem, padrão, transporte, ambiente heterogêneo, tudo isso nos remete ao XML, então vem a questão: Por que JSON e não XML? O principal motivo seria pelo fato da notação JSON ser extremamente mais simples do que a XML. Não existem tags diversas nem regras complexas para escrita e muito menos mecanismos complexos de parser.

No caso dos wizards para servidor a produtividade é ainda maior. Por exemplo, neste artigo utilizaremos a tecnologia REST para criar nosso servidor de aplicação. Com este wizard você tem condições de configurar vários parâmetros do servidor. Primeiro você escolhe se deseja uma aplicação Stand Alone ou ISAPI Application, assim seu servidor poderá ser executado como uma aplicação Win32 no primeiro caso, ou dentro de um servidor Web como IIS no segundo caso. Poderá ainda configurar qual a porta o servidor irá utilizar além de adicionar ou não suporte a autenticação HTTP e determinar a classe base de onde irão herdar as classes que conterão os Server Methods. Tudo isso de forma simples e rápida.

É importante ressaltar que, dependendo do tipo de aplicação selecionada, o wizard poderá apresentar um ou outro passo a mais ou a menos. Outra coisa que vale comentar é a presença de um novo componente relativo à DataSnap que foi adicionado a VCL, trata-se do componente

### Nota do DevMan

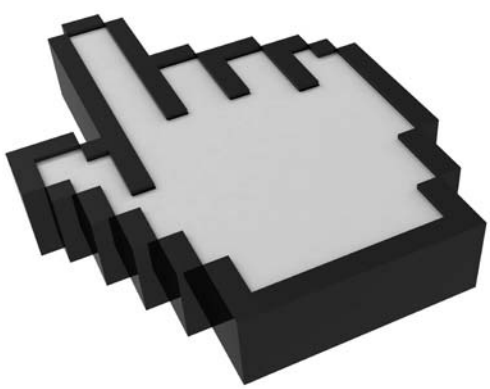
TDBXReader é uma classe do Framework DBExpress4 que fornece um leitor unidirecional para um conjunto de linhas de base de dados. TDBXReader é retornado por TDBXCommand. ExecuteQuery. O método TDBXReader.Next é utilizado para acessar a primeira linha e caso haja as sucessivas no ResultSet. Os valores da linha podem ser acessados usando a propriedade Value que é um array. A propriedade Value é sobrecarregada e podemos utilizar tanto a posição da coluna quanto seu nome.

### Nota do DevMan

São chamados Stand-Alone os programas completamente autossuficientes, ou seja, não necessitam de um software auxiliar, como um interpretador sob o qual terão de ser executados, para seu funcionamento. Por exemplo, um programa Java é tipicamente compilado para Bytecode e necessita de uma Java Virtual Machine para ser executado. Já um programa em Delphi depois de compilado não requer nenhum requisito para funcionar, exceto DLL de terceiros ou acesso a dados bem como pacotes quando aplicável.

### Nota do DevMan

Internet Server Application Programming Interface — ISAPI é um protocolo usado por computadores Windows para rodar uma aplicação dinâmica dentro de um processo em um servidor Web. Para usar o ISAPI, é necessário criar uma DLL que exporte alguns símbolos chamados pelo WebServer. Essas DLLs são carregadas na memória da máquina quando se inicia o servidor IIS, disponibilizando-as para o cliente.



## Nota do DevMan

Um proxy, em sua forma mais geral, é uma classe que funciona como uma interface para outra classe. Em padrão de projeto a função de uma classe proxy é prover um representante para outro objeto de modo a controlar o acesso a este. No caso do DataSnap podemos dizer que temos um proxy remoto, que é quando a classe proxy provê um representante local para um objeto em um espaço de endereçamento diferente, neste caso o servidor de aplicação.

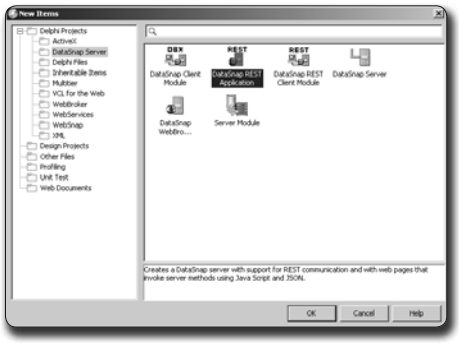


**Figura 1.** Novos wizards

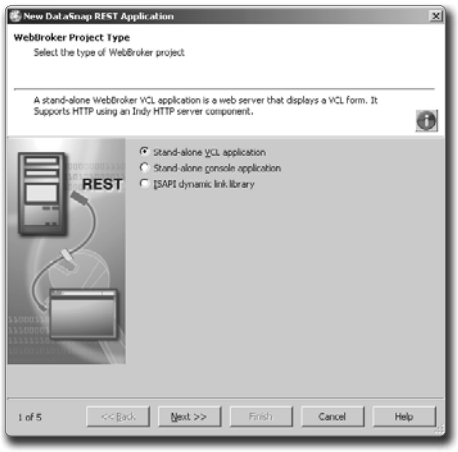


**Figura 2.** Escolhendo o tipo de aplicação

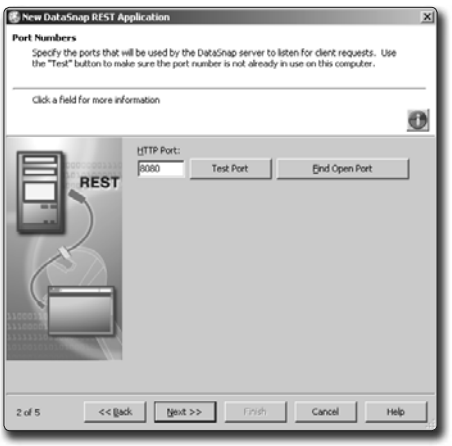


**Figura 3.** Selecionando a porta do servidor

DSProxyGenerator. Na nova arquitetura DataSnap lançada no Delphi 2009, os métodos do servidor ou Server Methods, podem ser acessados pelas aplicações clientes através de classes proxys, ou seja, uma unit é criada do lado cliente e nela são adicionadas classes com métodos para acessar as funções no servidor. Esse proxy até então era gerado em linguagem Delphi para Win32 e no Delphi 2010 foi adicionado suporte a geração de proxy para .NET. Agora com este novo componente podemos gerar proxy também para arquitetura REST. Isso mesmo, este componente tem a possibilidade de gerar um arquivo Javascript com funções para invocarmos os métodos no servidor via requisição HTTP.

Para que possamos ver na prática toda a facilidade que a arquitetura DataSnap nos proporciona, vamos desenvolver um pequeno exemplo onde teremos um servidor de aplicação Win32 baseado em REST. Vamos utilizar o novo wizard para criar o servidor. Além dos métodos que o wizard irá criar por padrão, vamos adicionar mais um método, chamado Login. Apesar se ser um regra simples de autenticação, a ideia central é mostrar em primeiro lugar como trabalhar com os novos wizards e entender o que são e para que servem cada um dos arquivos criados. Em segundo lugar visualizarmos como esta regra ou este serviço de login poderá ser consumido utilizando

## Nota do DevMan

As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, que é executado pelo compilador antes da execução do processo de compilação propriamente dito. Portanto, o pré-processador modifica o programa fonte, entregando para o compilador um programa modificado. Todas as diretivas de compilação no Delphi seguem o padrão de sintaxe {$DIRETIVA}. As diretivas podem ser colocadas em qualquer parte do programa. {$IFDEF VERXX} é um exemplo de diretiva que verifica a versão do compilador do Delphi onde a aplicação está sendo compilada.

**Nota**

É importante ressaltar que a porta utilizada no desenvolvimento pode não ser a mesma utilizada na produção. Por isso em uma aplicação real é interessante deixar esta configuração de forma parametrizada.

funções Javascript. Veremos ainda como registrar nossos métodos na lista de proxy para que estes métodos sejam adicionados às páginas HTML e assim possam ser consumidos via AJAX utilizando métodos em Javascript.

Nosso método login não terá nenhuma regra complexa. Ele receberá dois parâmetros, login e senha, e estes parâmetros serão submetidos a uma regra de validação. Sendo válido o usuário e a senha o método retorna true, caso contrário retornará false. Novamente o objetivo principal não é a metodologia de autenticação e sim como disponibilizar o serviço de login para uma aplicação seja Win32 ou Web. Será visto como criar um template HTML para ser usado em conjunto com o componente TPageProducer, podendo assim ver o método login em execução através de um template HTML sendo invocado por um Web Action.

### Criando o servidor com wizard

O ponto de partida do exemplo é o servidor de aplicação, para isso é usado o novo wizard (**Figura 1**). O wizard é composto por cinco passos a serem executados. No primeiro passo é escolhido o tipo de aplicação a ser criada (**Figura 2**). Isso faz com que seja criada uma aplicação estilo executável sem a necessidade de se instalar em um servidor Web como no caso do ISAPI, com a possibilidade de adicionar formulários VCL para criação de interfaces com o usuário, diferente de uma aplicação Standalone console onde há apenas interface em modo texto. A desvantagem nesse caso é que se perde todas os recursos oferecidos pelo servidor IIS.

No próximo passo é necessário determinar a porta que o servidor de aplicação utiliza para que as aplicações clientes possam consumi-lo. Por padrão a porta 8080 é indicada, porém ela pode estar sendo utilizada por outra aplicação na máquina onde o servidor irá ser instalado ou executado. A **Figura 3** ilustra a possibilidade de testar a porta escolhida além de obter uma porta vaga para uso.

A seguir, é tratada a possibilidade de adicionar ou não suporte a autenticação (**Figura 4**). A autenticação trata do acesso

ao servidor e garante que somente aplicações clientes que estejam devidamente autenticadas possam consumir os métodos. Além disso, ainda é possível determinar se será ou não adicionada uma classe container para os Server Methods, bem como se serão adicionados métodos de exemplo. Neste caso não será optado por trabalhar com autenticação HTTP.

O próximo passo trata da classe base para a Server Methods Class. Na nova arquitetura DataSnap para que seja possível exportar um método este deve estar em uma classe que herde de TDSProviderModule ou então herde de outra classe desde que tenha as diretivas {$MethodInfo ON} e {$MethodInfo OFF}. Isso é necessário para que os métodos que venham ser adicionados ao Server Methods Class possam ser lidos pelo mecanismo de reflexão do Delphi e assim possam, entre outras coisas, ser acessadas pelos clientes.

Na **Figura 5** é possível escolher se deseja herdar de TComponent, TDataModule ou TDSServerModule. A diferença entre eles é que herdando de TComponent não há como adicionar componentes na Server Methods Class, já herdando de TDataModule isto será possível. Agora herdando TDSServerModule há mais um recurso que é a implementação da interface IAppServer, permitindo assim que os DataSetProviders que venham a ser adicionados a ele possam ser mapeados pelos clientes.

A **Figura 6** ilustra o diretório aonde os arquivos do projeto serão gerados.

## Entendendo o servidor

No projeto criado foram adicionados um formulário, uma unit e um WebModule. O formulário tem a função de prover a interface principal da aplicação, nele existem alguns controles que são adicionados pelo Wizard e alguns métodos implementados. A unit possui em seu conteúdo a classe TServerMethods1 que serve como repositório para os métodos que serão exportados pelo servidor de aplicação, como definido no wizard a classe herda de TComponent e possui as diretivas de compilação {$MethodInfo ON} e {$MethodInfo OFF}, além do fato de possuir dois métodos de exemplo. Por fim o WebModule, que serve de repositório para os componentes que efetivamente transformarão a aplicação em um Servidor REST DataSnap.

Para facilitar o entendimento do WebModule a **Tabela 1** mostra a função de cada um dos componentes nele contidos e que são mostrados na **Figura 8**.

Além dos componentes, o wizard insere alguns métodos já implementados no Web Module, um dos principais é o método ServerFunctionInvokerHTMLTag. Na verdade ele é a implementação do evento OnHTMLTag dos componentes ReverseString e ServerFunctionInvoker. Este evento

### Nota do DevMan

As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, que é executado pelo compilador antes da execução do processo de compilação propriamente dito. Portanto, o pré-processador modifica o programa fonte, entregando para o compilador um programa modificado. Todas as diretivas de compilação no Delphi seguem o padrão de sintaxe {$DIRETIVA}. As diretivas podem ser colocadas em qualquer parte do programa. {$IFDEF VERXX} é um exemplo de diretiva que verifica a versão do compilador do Delphi onde a aplicação está sendo compilada.

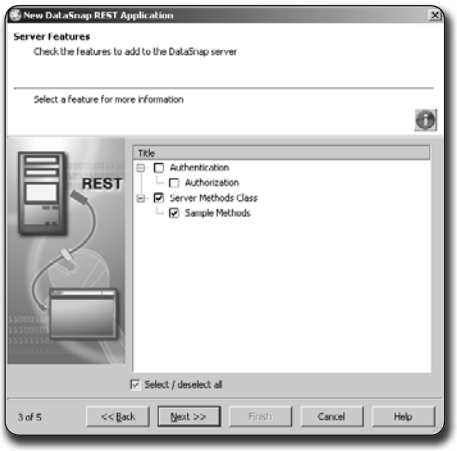



**Figura 4.** Adicionando Server Methods Class

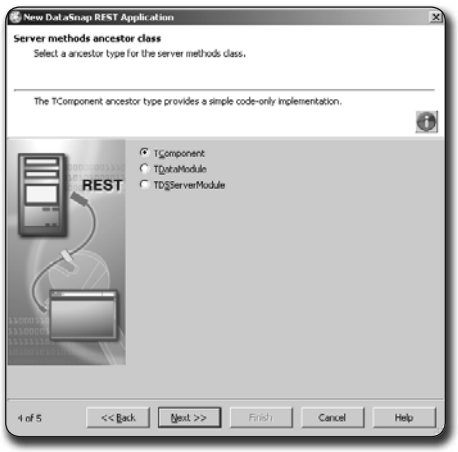



**Figura 5.** Selecionando classe base

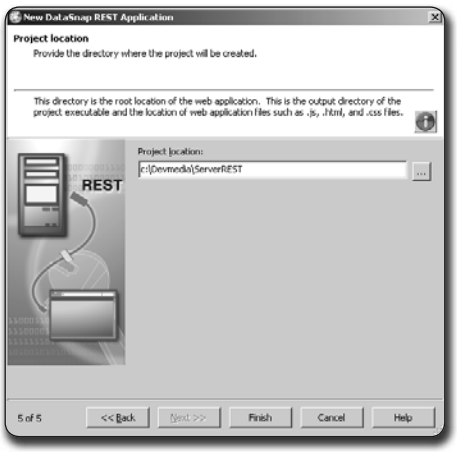



**Figura 6.** Localização do projeto

### Nota do DevMan

WebModule é o personagem principal do servidor, ele é quem recebe as requisições e executa os procedimentos necessários. A classe TWebModule não é novidade no Delphi, ela existe desde as versões anteriores do Delphi e é usada quando se quer desenvolver um Web Server Win32. O WebModule serve como um repositório de componentes não visuais, como TPageProducer, TSQLDataSet ou TDataSetTableProducer, além de permitir que a aplicação Web possa responder à requisições HTTP, passando essas requisições para ações apropriadas. O objeto WebModule gerencia uma coleção de ações, chamada WebActionItens que podem ser acessadas com um duplo clique no WebModule (**Figura 7**).

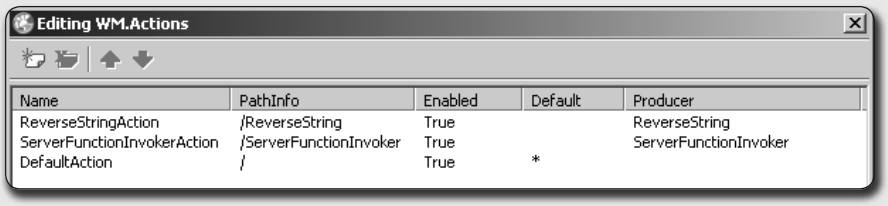



**Figura 7.** Ações do Web Module

| Componente | Função |
|---|---|
| DSServer1 | Componente que transforma uma aplicação em um servidor. |
| DSServerClass1 | Componente que "expõe" uma classe para que a mesma possa ter seus métodos consumidos pela aplicação cliente. Somente classes que descendem de TPersistent e possuam as diretivas {$MethodInfo ON} e {$MethodInfo OFF} em sua estrutura. |
| DSHTTPWebDispatcher1 | Componente responsável por receber as requisições feitas ao servidor DataSnap. |
| DSServerMetaDataProvider1 | Componente intermediário entre um DSServer e um ProxyGeneration que provê um MetaDataProvider para geração dos Proxys. |
| DSProxyGenerator1 | Componente responsável por gerar as classes proxys para se conectar e consumir um servidor de aplicação. |
| WebFileDispatcher1 | Componente que gerencia a requisição aos arquivos e diretórios que compõe o servidor. |
| ReverseString | Componente do tipo Page Producer que serve para trabalhar com templates HTML. |
| ServerFunctionInvoker | Componente do tipo Page Producer que serve para trabalhar com templates HTML. |

**Tabela 1.** Componentes do WebModule

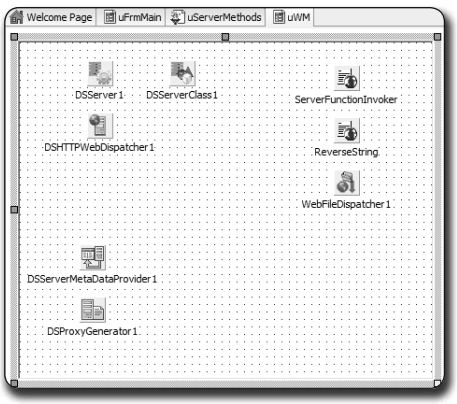



**Figura 8.** Componentes no WebModule

**Listagem 1**. Declarando o método login

```
{$METHODINFO ON}
 TServerMethods1 = class(TComponent)
 private

 public
  function EchoString(Value: string): string;
  function ReverseString
    (Value: string): string;
  function Login(strUser, StrPass: String):
    boolean;
end;
{$METHODINFO OFF}
```

é chamado quando o template ao qual estes componentes estão ligados são carregados. Desta forma, as tags nos templates são substituídas por qualquer texto passado neste evento. Por exemplo:

```
setConnection('<#host>','<#port>','<#urlpath>');
```

Aqui existem três tags que precisam ser substituídas por valores e isto é feito no evento OnHTMLTag, repare como o método está implementado, ele verifica o nome da tag que está sendo processada no momento e redefine o conteúdo da

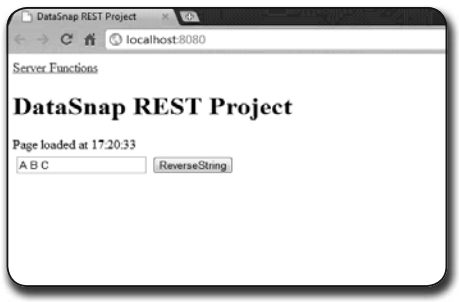



**Figura 9.** Servidor em execução

tag por uma propriedade do servidor de DataSnap. Outro exemplo:

```
...
if SameText(TagString, 'urlpath') then
  ReplaceText := string(Request.InternalScriptName)
else if SameText(TagString, 'port') then
  ReplaceText := IntToStr(Request.ServerPort)
else if SameText(TagString, 'host') then
  ReplaceText := string(Request.Host)
...
```

Se verificar o diretório da aplicação notará que foram gerados vários arquivos organizados em pastas, existem arquivos CSS para o layout dos templates, algumas imagens e principalmente scripts, que possuem todos os métodos para interagir com o servidor de aplicação apenas utilizando Javascript.

A aplicação pode ser executada de dentro do Delphi XE, processo que inicializa o servidor. A opção Open Browser abre o navegador para que o servidor seja testado. A ação default do WebModule é invocada e irá chamar o template ReverseString.html, o evento OnHTMLTag será invocado, as tag carregadas e o resultado final será exibido no browser (**Figura 9**).

Na **Figura 9** há um link logo acima chamado Server Functions, esse link irá levar para uma página que exibe todos os métodos disponíveis no servidor e ainda permite executar cada um deles. O mais interessante é que ele monta uma interface HTML dinamicamente para que seja possível invocar estes métodos, isso é muito bom, pois permite testar os métodos por criados sem que tenha que investir tempo para criar uma interface (algo como faz o .NET para Web Services). **A Figura 10** mostra a página com as Server Functions sendo exibidas.

## Novo método no servidor

O método do exemplo irá efetuar um pseudo login, a intenção na verdade é aprender a adicionar este novo método na lista de proxy e em seguida invocá-lo via AJAX através de métodos Javascript. O método chamado Login está na classe TServerMethods1 (**Listagem 1**).

O método login como de praxe solicita o usuário e a senha e a sua implementação é bem simples:

```
Result := strUser = StrPass;
```

Como o novo método adicionado no Server Methods, este já pode ser consumido por uma aplicação cliente qualquer. Para testar o método, ele deve ser disponibilizado para que possa ser exibido na lista de métodos, sendo consumido pela interface HTML que é gerada dinamicamente. É preciso adicionar o novo método na lista de métodos no arquivo Javascript. No arquivo ServerFunctions.js dentro da pasta js no diretório da aplicação, deve ser alterada a variável JSProxyClassList e no array TServerMethods1 adicionado o método login conforme a seguir:

```
"TServerMethods1": ["EchoString",
  "ReverseString","Login"]
```

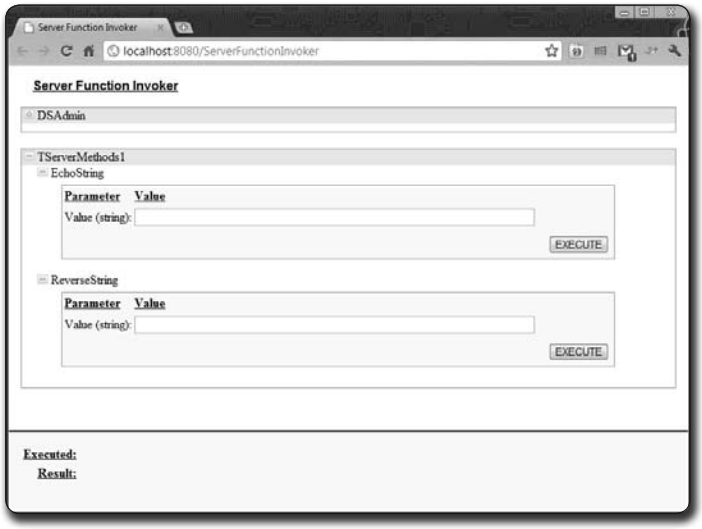



**Figura 10.** Exibindo a página com as Server Functions

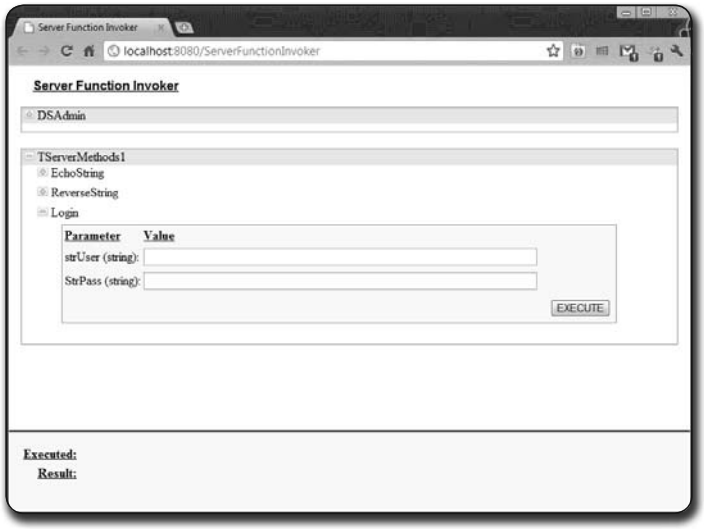



**Figura 11.** Exibindo o método Login

Isso é o suficiente para testar o método utilizando o template criado pelo wizard. A **Figura 11** exibe a página com os Server Functions e agora o método Login já aparece na lista de métodos. Na mesma figura é possível perceber a criação de um input text para cada parâmetro do método login, essa infra já está implementada em um método Javascript disponível no arquivo ServerFunctionInvoker.js chamado createClassTree. Ele é quem faz todo o trabalho pesado.

Agora não somente criar um servidor baseado em REST ficou fácil, consumí-lo também está extremamente simples. Os arquivos Javascript que foram gerados pelo wizard e estão na pasta "js" possuem dezenas de métodos que automatizam o consumo dos métodos no servidor. Para entender melhor basta criar um novo template (**Listagem 2**) para consumir o método Login.

**Listagem 2**. Template Login HTML

```
01 <html>
02 <head>
03    <title>DataSnap REST Server - Login</title>
04    <script type="text/Javascript" src="js/json-min.js"></script>
05    <script type="text/Javascript" src="js/serverfunctionexecutor.js"></script>
06    <script type="text/Javascript" src="js/connection.js"></script>
07    <script type="text/Javascript" src="/js/serverfunctions.js"></script>
08 </head>
09 <body onload="onLoad()">
10    <h1>DataSnap REST Server</h1>
11    <div id="contentdiv" class="contentdiv">
12     <table>
13      <tr><td>Usuário: <input id="UserValue" type="text"/></td></tr>
14      <tr><td>Senha: <input id="PassValue" type="text"/></td></tr>
15      <tr><td><button onclick='Javascript:onLoginClick();'>Login</button></td></tr>
16     </table>
17    </div>
18 </body>
19 </html>
```

A **Listagem 2** exibe um código HTML simples onde há no conteúdo da página uma tabela com três linhas e uma coluna. Cada linha exibe um TextBox para username, um segundo para a senha e um botão respectivamente. Alguns detalhes precisam de explicação, por exemplo, nas linhas 4 a 7 os códigos que fazem referência a alguns dos arquivos Javascript que foram criados pelo wizard. Repare que são carregados os arquivos da pasta "js", pois quando a página Login.html é executada, é carregada como se estivesse no diretório do executável, uma vez que ela é chamada pelo WebModule.

Outros dois detalhes importantes são: o evento onLoad da tag body está apontando para uma função Javascript chamada onLoad(), na linha 9. Essa função é criada na sequência e tem um papel importante. O segundo detalhe encontra-se na linha 15 onde é apontado o evento onclick do elemento HTML Button para uma função Javascript chamada onLoginClick() que também é criada em seguida. Para concluir o template Login.html são adicionadas dentro das tags <head></head> as funções Javascript mencionadas. A **Listagem 3** mostra o código das funções.

A **Listagem 3** mostra três funções Javascript, a primeira delas é a função onLoad. Nesta função é invocado o método setConnection que está definido no arquivo connection.js. Um detalhe é que as tags são passadas como parâmetros, essas tags são substituídas peles seus respectivos valores através do evento onHTMLTag que já está definido no WebModule. O segundo método chamado serverMethods() tem a função de criar um objeto do tipo da ServerClass da qual se quer consumir os serviços. Neste caso é consumido o método Login que está na Server Class TServerMethods1, assim a função do método serverMethods é criar um objeto do tipo TServerMethods1 para que através dele seja possível invocar os métodos disponíveis.

Por fim o método onLoginClick(), que como citado é chamado pelo evento onclick do botão HTML disponível no template. Este é o método que efetivamente invocará a função login no servidor DataSnap. Neste método são criadas duas variáveis chamadas UserValue e PassValue que recebem a referência dos controles TextBox. Na sequência

o método serverMethods() é utilizado para chamar a função Login, para esta função são passados os valores do usuário e senha que foram digitados nos respectivos TextBoxs. O resultado da função Login é armazenado na variável "s" e em seguida invocado o método alert() do Javascript para exibir o resultado para o usuário.

**Nota**

O método alert() do Javascript tem a mesma função do método ShowMessage no Delphi. Ele exibe uma caixa de diálogo com o texto passado como parâmetro.

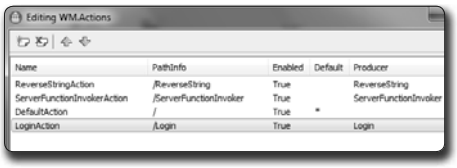



**Figura 12.** WebActions Editor

**Figura 13.** Método Login em execução

**Listagem 3**. Funções Javascript para conexão com servidor

```
<script type="text/Javascript">
function onLoad()
{
  setConnection('<#host>', '<#port>', '<#urlpath>');
}

function serverMethods()
{
  return new <#classname>(connectionInfo);
}

function onLoginClick()
{
  var UserValue = document.getElementById('UserValue');
  var PassValue = document.getElementById('PassValue');
  var s = serverMethods().Login(UserValue.value, PassValue.value);
  alert(s.result);
}
</script>
```

Para finalizar o exemplo resta adicionar o novo template ao servidor de aplicação. Para isso é adicionado mais um componente TPageProducer ao WebModule. Aqui basta alterar sua propriedade HTMLFile para o template Login.html. Deve-se aqui utilizar o caminho relativo, ou seja, apenas templates\Login.html, com isso o servidor irá saber que deve carregar o arquivo de um subdiretório templates não importando o diretório onde o servidor esteja rodando. A segunda alteração é apontar o evento OnHTMLTag para o método ServerFunctionInvokerHTMLTag que já está pronto e implementado. Uma vez tendo configurado o template basta criar um novo WebAction para que possa invocar este novo template. Basta criar um novo WebActions e apontar sua propriedade Producer para o componente Page Producer Login. A **Figura 12** exibe os WebActions devidamente configurados.

O teste pode ser feito na mesma opção Open Browser vista anteriormente, que irá chamar o browser e exibirá a página default. Na barra de endereços agora pode-se acessar o endereço http://localhost:8080/Login, que irá invocar o novo template. A **Figura 13** mostra o método em execução, lembrando que o método retorna true se ambos usuário e senha forem idênticos.

## Conclusão

O que foi mostrado aqui é muito pouco perto de todo o poder que a arquitetura DataSnap oferece. Esta arquitetura não é nova, está presente no Delphi desde a versão 7 de forma mais estável. Com as mudanças sofridas nas últimas versões elevou-se e muito o nível da arquitetura.

O método login mostrado aqui é apenas um aperitivo, como você pode perceber utilizamos apenas HTML e Javascript para consumir os recursos do servidor. Isso significa que podemos utilizar qualquer linguagem para desenvolver aplicações clientes baseada na Web para consumir nosso servidor. Trafegar objetos, retornar ResultSets e muito mais, se torna uma tarefa fácil diante dos recursos apresentados aqui.

**Rodrigo Carreiro Mourão**
*rodrigocarreiro@rmfactory.com.br*
*rodrigocarreiro.com.br*
*twitter.com/carreiromourao*
*radphp.com.br*
Sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria e desenvolvimento utilizando soluções RAD da Embarcadero. Instrutor certificado pela Borland e Delphi Product Certified. Editor técnico da revista Clube Delphi, administrador da Comunidade RAD PHP, membro ativo na comunidade e apaixonado pelo Delphi.

**Download Versão Trial RAD Studio XE**
*http://www.embarcadero.com.br/products/rad-studio*

**Documentação Online sobre WebModule**
*http://docwiki.embarcadero.com/RADStudio/en/The_Web_Module*

**Dê seu feedback sobre esta edição!**

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

***www.devmedia.com.br/clubedelphi/feedback***

# Estenda seu Delphi XE

## Aprenda a criar experts e plug-ins

Vitor Luiz Rubio

## Resumo DevMan

***De que trata o artigo:***

Neste artigo será mostrado como usar a OTA (Open Tools API) para criar experts. Experts são extensões e plug-ins para o Delphi, que vão desde editores de código a organizadores de componentes.

***Para que serve:***

Experts são extensões para Delphi com os mais variados objetivos. Eles visam diminuir o tempo que se gasta com tarefas repetitivas, como renomear componentes, trocar toda uma classe de componentes, organizar e formatar código, dar check-in ou check-out em um repositório de uma ferramenta de versionamento, backup dos fontes entre outros.

***Em que situação o tema é útil:***

Sempre que existir a necessidade de fazer alguma tarefa repetitiva pode existir algum expert que auxilie. Se não existir pode ser feito um. Pode-se inclusive tornar o Delphi muito mais útil por meio de experts e vender assim produtos baseados em experts comerciais.

***Estenda seu Delphi XE:***

Este artigo abordará o básico de experts: quais as interfaces devem ser usadas, como implementá-las, como criar, como debugar, e como tornar o desenvolvimento de software mais ágil com a ajuda deles. Serão mostrados alguns experts para agilizar tarefas rotineiras do nosso dia – a – dia. Com isso você poderá não só criar experts para Delphi, mas para C++ Builder também. Eles auxiliam a economizar tempo e dinheiro automatizando tarefas, como deploy por exemplo.

Alguns experts estão tão consolidados na comunidade Delphi que se tornaram essenciais. Quem hoje ficaria sem o GExperts, ou o CNExperts? Sem contar os vários plug-ins que já vêm com o próprio Delphi, por exemplo a versão XE já vem com um plug-in para integração com o SVN.

Trabalhar com experts envolve criar bibliotecas (DLL ou BPL) que serão usadas pelo próprio Delphi e podem interagir com quase qualquer janela do Delphi, desde as janelas de código até as de edição de formulário. Para desenvolver essas bibliotecas é necessário implementar algumas interfaces que são pouco conhecidas e pouco documentadas do Delphi. É possível encontrar na internet material a respeito da OTA (Open Tools API), mas sempre em inglês, desatualizado e fragmentado.

Este artigo não visa documentar toda a OTA porque muitas páginas seriam necessárias para isso. O objetivo deste é apenas introduzir o leitor nas técnicas de criação de complementos para o IDE fazendo um extenso uso de interfaces, packages e POO. Para que não seja muito extenso e enfadonho assume-se que o leitor já está familiarizado com esses assuntos.

É um pré-requisito para o entendimento desse artigo um conhecimento sólido em criação de pacotes e BPLs e em interfaces e POO. Outro pré-requisito importante é o uso da versão Architect do produto, pois as versões inferiores não possuem o código fonte da ToolsAPI.pas.

É necessário compreender os perigos de criar partes do IDE do Delphi. Um acesso a um objeto destruído ou qualquer erro causará instabilidade. Por isso existe a necessidade de manter-se um bom programa de backup e / ou controle de versão e de se ativar o auto-save para as units. Há uma lista de interfaces que usaremos neste artigo, mas a Open Tools API tem muitas outras. Ler o código da unit ToolsAPI.pas e os comentários será essencial para seu aprendizado.

Ao se implementar várias interfaces muitos métodos necessitarão apenas existir, não precisando ser implementados com um corpo. Isso porque o Delphi pode já ter outras classes de wizards e plug-ins já registrados que implementam esses métodos e fazem alguma coisa. Além disso, muitas funções devem retornar string vazia ou qualquer outro valor padrão para que o Delphi decida automaticamente qual valor usar. Certos métodos são implementados apenas em wizards mais complexos.

Qualquer wizard ou expert para o Delphi deve ser registrado. Quando isso ocorre um objeto da classe que você criou é adicionado em uma lista de objetos do mesmo tipo ou interface. Isso quer dizer que nem todos os experts que você escrever substituirão funções do Delphi. Alguns se agregarão e executarão antes ou depois das funções nativas do Delphi, e mesmo assim, dependendo do tipo de Wizard, você pode escolher se o nativo do Delphi será usado ou não.

### Nota do DevMan

O padrão Observer garante que as partes específicas de um sistema sejam notificadas de mudanças que acontecem em alguma área do sistema, porém, sem que estas duas partes saibam muito uma da outra.

### Nota do DevMan

Package – É uma biblioteca de link especial do Delphi. Elas possibilitam que partes de códigos, ou rotinas, sejam separadas em módulos que podem ser compartilhados com várias outras aplicações. Elas também podem ser utilizadas como meio de instalação de componentes de terceiros. Basicamente existem dois tipos de packages: runtime e design-time. As de runtime disponibilizam seus recursos durante a execução do projeto, já as outras são usadas para instalar componentes e criar editores de propriedade diferenciados.

### Nota do DevMan

Package vs. DLL - Já que ambos são utilizados para armazenar código que pode ser compartilhado, qual a diferença entre esses dois tipos de arquivo? Os packages são específicos do Delphi, isto é, programas feitos em outras linguagens não podem reutilizar um pacote BPL. É para isso que existem as DLLs. De uma forma geral uma DLL contém uma coleção de rotinas, já uma BPL pode conter units completas, componentes, formulários, mas também classes, permitindo assim código orientado a objetos. O que a BPLs podem fazer e as DLLs não, é conter um conjunto de componentes dentro de si.

### Nota do DevMan

Open Tools API, mais conhecido como OTA, é o nome dado ao conjunto de interfaces e funcionalidades pelo Delphi para a criação de experts, assistentes e plugins para o IDE. Seu código-fonte é distribuído com o Delphi, no diretório Source/ToolsAPI. A Embarcadero disponibiliza também as fontes de parte dos assistentes do IDE, alguns inclusive fazem uso de recursos do OTA. No diretório Source/Property Editors podemos encontrar o código de importantes partes do IDE, como o editor de TFields, o editor da propriedade CommandText do SQLDataSet, entre outros. Usando as interfaces do OTA, podemos fazer praticamente qualquer coisa no IDE: manipular o editor de código, barras de ferramentas, menus, o form designer, etc. Bons exemplos de uso avançado do OTA são os utilitários GExperts e CodeRush. Existe um personagem famoso na comunidade por promover o uso do OTA. Erick Berry possui uma página com excelentes exemplos de uso da tecnologia (visite www.gexperts.org/opentools). Erick foi um dos criadores do GExperts. Por seu excelente trabalho, ele também acabou dando uma "mãozinha" para a Borland na construção das interfaces OTA do IDE do BDS.

**Nota**

As units que trabalham com OTA devem ter a declaração ToolsAPI no seu uses. A unit ToolsAPI encontra-se escondida no pacote DesignIDE.dcp, por isso ela deve ser incluída no requires do dpk

Muitas coisas o Delphi faz sozinho, sem usar nenhum wizard, os wizards são apenas "pontos de entrada/comunicação" com as funções do Delphi.

É interessante que alguns métodos de alguns tipos de interfaces da OTA nunca serão chamados pelo IDE. Isso acontece porque esses métodos foram reservados para uma implementação futura e estão aguardando o dia em que a equipe de desenvolvimento do Delphi fará implementações no IDE que chamem esses métodos. Vale salientar que a OTA nunca foi completamente documentada e que está em evolução. A cada versão do Delphi ela muda um pouco.

Existem mais de 100 interfaces na OTA, e elas são divididas em vários grupos. Não serão abordados todos os grupos nesse artigo, apenas:

Services → utilidades gerais providas pelo IDE do Delphi;

Notifiers → seguem o padrão Observer, servindo para notificar um expert da inclusão ou exclusão de um componente num editor de form, ou o caminho inverso, notificar o form de uma mudança no expert;

Modules → todo artefato criado pelo desenvolvedor no Delphi é um módulo. Units são módulos de um arquivo, forms são módulos de dois arquivos;

Creators → factorys responsáveis por criar alguma coisa. Neste artigo são mostrados criadores de módulos.

Algumas das interfaces abordadas neste artigo são:

IOTAWizard → Sempre deve ser implementada por um wizard. Ela possui o método execute que é onde realmente a execução se dará. Precisa ser registrada no Delphi;

IOTAMenuWizard → Implementando-se essa interface é possível adicionar um item de menu debaixo do menu help;

IOTANotifier → Registra ou "observa" eventos ocorridos em uma classe e os repassa para outra onde ele esteja adicionado/registrado. É a base da maioria dos wizards;

IOTAFile → Base dos wizards para se trabalhar com arquivo.

Quando se fala de OTA geralmente se usam as terminologias experts ou wizards para tratar de plug-ns para o Delphi, qualquer que seja sua natureza. Por isso nem sempre um artefato chamado "wizard" será de fato uma janela de assistente com várias etapas de avançar e concluir no final, mas podem ser.

## Hello World

Para manter a tradição o expert mais trivial que pode ser feito é um wizard para exibir a mensagem "Hello World". Para se trabalhar com a OTA é necessário criar bibliotecas do tipo DLL ou packages do Delphi que gerem BPLs. É muito mais fácil o uso de packages, pois o uso de DLLs requer alterações no registro do Windows e são mais difíceis de se depurar. Com BPLs é possível instalar e desinstalar os plugins com o IDE aberto.

O exemplo da **Listagem 1** cria uma classe chamada TWHello que implementa as interfaces IOTAWizard e IOTAMenuWizard. O primeiro desafio enfrentado é a implementação da interface IOTAWizard, pois como ela é descendente de IOTANotifier, torna-se necessária a implementação de seus métodos também. Também é mostrada uma implementação básica de IOTANotifier com seus métodos apenas declarados, mas vazios. IOTANotifier é usado em vários tipos de wizards, principalmente os que fazem comunicação entre o IDE e uma outra parte do Delphi, para que uma parte seja notificada de eventos ocorridos na outra.

A Interface IOTAWizard pode enganar pelo nome: ela não serve simplesmente para criar wizards, mas virtualmente qualquer código criado para ela funcionaria sem problemas. Ela possui os métodos:

GetIDString: string – uma string única para identificar este plug-in dentre todos os outros do Delphi;

GetName: string – um nome completo, geralmente junto com o nome da empresa que criou o wizard;

GetState: TWizardState – Um estado que pode dizer se o wizard está habilitado ou não e se ele é do tipo "checável", e se está checado ou não;

Execute – O principal método, é aqui que a lógica toda vai ser colocada.

Também é possível fazer uma classe descendente de TNotifierObject, assim torna-se desnecessário implementar a interface IOTANotifier porque TNotifierObject já a implementa.

Sem aparecer em algum lugar do Delphi o wizard seria inútil, por isso deve-se implementar também a interface IOTAMenuWizard. Pode-se acessar o source dessas interfaces usando Ctrl + Click. Os comentários lá contidos são muito úteis para orientar desenvolvedores de plug-ins. Esta interface contém um único método que é GetMenuText. O exemplo Hello World deste artigo mostra como colocar o wizard para ser executado através de um menu chamado Hello Menu.

Tendo implementado todos os métodos, ou pelo menos seu esqueleto, é necessário uma procedure pública e estática fora do escopo da classe TWHello que é a procedure Register. Ela deve começar com "R" maiúsculo mesmo. O IDE procurará por ela e a executará automaticamente ao proceder-se a instalação do pacote, e isso fará com que expert seja instalado e comece a funcionar.

O construtor da classe contém o código ShowMessage('Criando o Hello Wizard') apenas para se observar o momento exato em que a classe do wizard é instanciada. Essa parte pode ser removida porque não faz nada de útil e ainda pode atrapalhar - toda vez que o Delphi for aberto essa mensagem será disparada. Neste caso, fechando e abrindo o Delphi será exibida a mensagem 'Criando o Hello Wizard' durante a splash screen. A package uma vez compilada e instalada, instantaneamente o Delphi exibirá a mensagem de criação (ocorre apenas uma vez) e no menu Help aparecerá o sub menu 'Hello Wizard'.

Uma alternativa interessante a esse código seria fazer com que a classe TWHello fosse descendente de TNotifierObject em vez de TInterfacedObject, poupando algumas linhas de código. Os próximos exemplos serão todos descendentes de TNotifierObject. Olhando o fonte de TNotifierObject vê-se que ele também não implementa realmente a interface, mas deixa o esqueleto dos métodos em branco para que possam ser sobrecarregados pelos seus herdeiros.

A **Figura 1** mostra o menu que foi adicionado. Clicando-se nele simplesmente vê-se uma mensagem de "Hello World".

## Experts com Menu

Se o objetivo não é que o expert fique no menu Help (o que não faz o menor sentido) saiba que isso é simplesmente o padrão, mas é possível escolher onde o menu aparecerá. Para isso é necessário criar alguns TMenuItens e adicioná-los ao menu do

**Nota**

Observe que a procedure Register para experts é ligeiramente diferente do registro de componentes.

**Nota**

É uma boa prática colocar em todas as units de Experts um prefixo seu que seja único para que a unit criada por você em seu pacote não tenha conflitos de nomes com units internas do Delphi.

**Listagem 1**. Wizard Hello World

```
unit uHello;

interface

uses Dialogs, ToolsAPI;

type
 TWHello = class(TInterfacedObject, IOTAWizard, IOTAMenuWizard )
 public
  //implementação de IOTAWizard
  function GetIDString: string;
  function GetName: string;
  function GetState: TWizardState;
  procedure Execute;
  //implementação de IOTAMenuWizard
  function GetMenuText: string;
  // implementação de IOTANotifier
  // Não implementaremos esses métodos
  procedure AfterSave;
  procedure BeforeSave;
  procedure Modified;
  procedure Destroyed;
  constructor Create;
 end;

procedure Register;

implementation

function TWHello.GetIDString: string;
begin
 Result := 'ClubeDelphi. Wizard Exemplo';
end;

function TWHello.GetName: string;
begin
 Result := 'Hello Wizard';
end;

procedure TWHello.Execute;
begin
   ShowMessage('Hello World');
end;

function TWHello.GetState: TWizardState;
begin
 Result := [wsEnabled];
end;

function TWHello.GetMenuText: string;
begin
 Result := 'Hello Menu';
end;

constructor TWHello.Create;
begin
 inherited;
 ShowMessage('Criando o Hello Wizard');
end;

procedure Register;
begin
 RegisterPackageWizard(TWHello.Create);
end;

end.
```

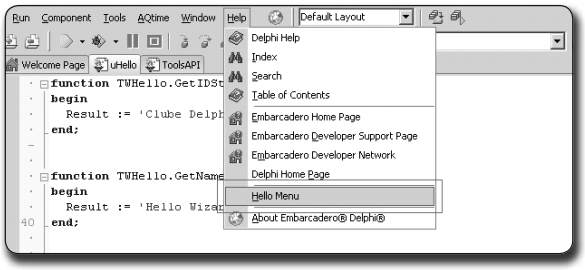



**Figura 1.** Hello Menu

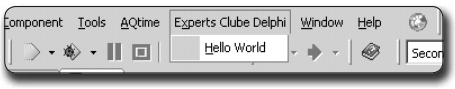



**Figura 2.** Menu de um wizard

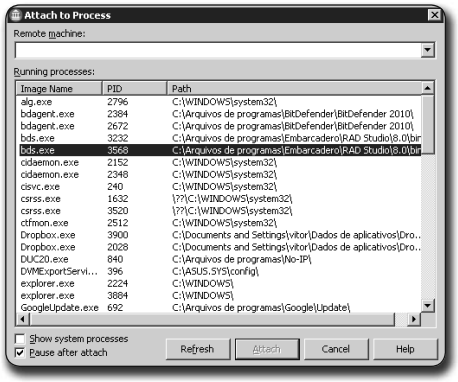



**Figura 3.** Anexando o processo ao debugger

**Listagem 2**. Evento Click do menu

```
procedure TWHello.MenuHelloClick(Sender:
TObject);
begin
  Execute;
end;
```

**Listagem 3**. Construtor e Destrutor

```
constructor TWHello.Create;
begin
  inherited;
  FRaizNossoMenu := TMenuItem.Create(nil);
  FNossoMenu := TMenuItem.Create(nil);
  FNossoMenu.OnClick := MenuHelloClick;
  FRaizNossoMenu.Caption :=
    'Experts Clube Delphi';
  FNossoMenu.Caption := 'Hello World';
  FRaizNossoMenu.Add(FNossoMenu);
  (BorlandIDEServices as INTAServices).
    MainMenu.Items.
  Insert(10, FRaizNossoMenu);
end;

destructor TWHello.Destroy;
begin
  FNossoMenu.Destroy;
  FRaizNossoMenu.Destroy;
  inherited;
end;
```

IDE, escolher entre adicionar um submenu a um menu já existente ou adicionar desde a raiz. Um exemplo interessante é o GExperts, que tem uma opção para adicionar seus menus debaixo de Tools mas que também pode ser configurado para ser acessível desde o menu principal, sob o nome GExperts.

Para isso basta colocar na seção private da classe dois objetos do tipo TMenuItem (adicionando a unit Menus no uses) e criar esses menus no Create do TWHello e Destruir no Destroy. Para o menu ser clicado deve-se ter uma procedure com assinatura de evento que chame o método Execute. A **Listagem 2** mostra a procedure com assinatura de evento "Click" (com parâmetro Sender) e a **Listagem 3** mostra o construtor e o destrutor do TWHello.

É possível criar inúmeros menus e inúmeros eventos OnClick correspondentes a cada um deles. Também é possível listar e varrer todos os nomes de menus e submenus para adicionar um em qualquer lugar. Cada evento poderia realizar sua própria tarefa que não necessariamente poderia ser o método Execute.

Dessa forma não seria necessário implementar a interface IOTAMenuWizard. Assim o menu não ficará debaixo do Help, mas pode ser colocado arbitrariamente em qualquer lugar.

No exemplo o menu foi propositalmente colocado na posição 10. O resultado dependerá muito de quantos menus ou plugins a instalação usada do Delphi já possua. Na **Figura 2** o menu de exemplo foi colocado à esquerda do menu Window.

## Debugando

Para depurar um expert é necessário o mesmo método usado para depurar componentes e DLLs: é necessário que um executável que use a biblioteca seja anexado ao processo que deseja-se depurar. A diferença é: qual processo vai utilizar um expert do Delphi? O próprio Delphi. Por isso outra instância do IDE é usado para fazer o debug. Deve-se abrir o gerenciador de tarefas e verificar qual é a instância do Delphi que está sendo usada para programar e qual é a segunda que foi aberta. Depois disso se faz o "attach" do processo como mostra a **Figura 3**. Anexa-se o bds.exe de número de PID diferente do PID da instância de desenvolvimento, ou seja, o segundo que foi aberto.

A depuração iniciará pausada, mas ao clicar no Run (com debug) ela entra em execução. Agora pode-se acompanhar o passo a passo do código. Para se ver a depuração em execução é necessário inserir um breakpoint no ShowMessage('Hello World') da instância que está em uso para desenvolvimento, e na segunda instância, através do menu Help>Hello Menu dispara a execução do expert. Assim o código que é executado fica parado no breakpoint, possibilitando o uso do Step Into, Step Over e outras opções do Debugger. Para tornar o processo mais fácil pode-se colocar nas opções de Run > Parameters o próprio Delphi como Host Application.

## Criando um arquivo

A criação de um arquivo envolve a implementação de interfaces que interajam com o repositório. A interface IOTARepositoryWizard deve ser implementada para que a integração com o repositório seja possível. Entretanto essa interface possui dois descendentes que não acrescentam nenhum método novo: IOTAProjectWizard e IOTAFormWizard. Estes descendentes atuam apenas como flags de tipo. Para criar um assistente de repositório que funcione sempre, logo que se abre o Delphi, usa-se IOTAProjectWizard e para criar um assistente que funcione apenas quando um projeto que use forms estiver aberto implementa-se o IOTAFormWizard.

O wizard do repositório por sua vez fará uso de um IOTACreator, mais especificamente de um de seus descendentes, IOTAModuleCreator. IOTAModuleCreator é uma interface usada para criar módulos. Módulos são as classes que implementam IOTAModule (e seus descendentes) e podem ser qualquer arquivo ou parte formada por vários arquivos tratada pelo Delphi.

Uma unit é um tipo de módulo formado apenas por um arquivo .pas. Um formulá-

rio é um módulo formado por dois arquivos, um .pas e um .dfm. Pacotes, projetos e Project groups também são módulos.

Por último existe a interface IOTAFile responsável pela criação e manutenção do arquivo fisicamente. Um ModuleCreator vai delegar a tarefa de criar um arquivo a um IOTAFile.

Esse desenvolvimento se dá em três camadas:

- A implementação de IOTAProjectWizard e / ou IOTAFormWizard;
- A Implementação de IOTAModuleCreator;
- A Implementação de IOTAFile.

Essas três interfaces podem ser implementadas por apenas uma classe, contanto que existam todos os métodos. Mas o ideal é que se crie uma classe para cada interface, assim se reduz acoplamento e as classes podem ser trocadas ou integradas com outras classes.

O próximo exemplo mostra a criação de um arquivo do tipo unit. Para adicionar uma unit no projeto é preciso varrer a lista de módulos carregados pelo IDE (cada arquivo aberto é um módulo) para poder obter o arquivo do Project Group (sempre existe, mesmo que não seja salvo ou que só exista um projeto). Com o módulo do Project Group pode-se obter o módulo do projeto ativo e assim saber-se onde adicionar o novo módulo a ser criado.

Para isso foi criada uma unit chamada uUtilidades que contém alguns métodos estáticos globais para auxiliar na tarefa de obter o Project Group e o projeto ativo. Essa unit pode ser encontrada na **Listagem 4.**

ProjectGroupAtivo é a unit que percorre a lista de módulos e verifica se algum desses módulos é um IProjectGroup, o tipo do módulo que trata de Project Groups. QueryInterface é o método que verifica se uma interface é de um determinado tipo ou se é compatível com uma variável.

A função ProjetoAtivo usará a ProjectGroupAtivo para obter o módulo de grupo de projetos e se retornar (diferente de nil) ele retornará o projeto ativo, caso contrário ele retornará nil também.

Para criar um arquivo é necessário implementar a unit IOTAFile que contém o conteúdo do arquivo, e depois criar uma classe que implemente IOTAModuleCreator. IOTAFile cria o conteúdo do arquivo e o próprio arquivo fisicamente, mas é IOTAModuleCreator que cria ou adiciona um novo módulo (unit) ao Delphi, usando IOTAFile para saber como criar um arquivo.

O fluxo é o seguinte: o Wizard implementador de IOTAWizard não sabe como criar um módulo, por isso instancia um "fazedor de módulos" (IOTAModuleCreator) para adicionar o módulo ao projeto, porém o próprio IOTAModuleCreator não sabe como "montar" um arquivo, por isso ele instancia um IOTAFile que é responsável por ter o mecanismo de criar um arquivo a partir de um template.

Veja como o conceito de Factory Method foi bem utilizado pela equipe que idealizou a Open Tools API. O wizard criado não tem conhecimento de como criar uma unit no projeto, por isso instanciará um criador de units que por sua vez não tem conhecimento de como criar um arquivo então delegará essa tarefa a um criador de arquivos.

A **Listagem 5** mostra a classe TCriadorArquivoUnit na unit uCriadorArquivoUnit cuja função é criar um arquivo do tipo unit. Dentro da unit encontra-se

### Nota do DevMan

Acoplamento - O conceito de acoplamento está relacionado com o grau de dependência que um objeto tem em relação a outro. Baixo acoplamento (loose coupling) significa que um objeto possui baixa dependência, já alto acoplamento (tight coupling) indica que a dependência é grande. Quanto mais baixo o acoplamento, mais fácil é alterar a codificação de uma classe, visto que você não precisa alterar os demais objetos que dependem do primeiro.

**Listagem 4.** Obtendo o projeto corrente

```
unit uUtilidades;

interface

uses
  ToolsAPI;

function ProjetoAtivo: IOTAProject;
function ProjectGroupAtivo: IOTAProjectGroup;

implementation

function ProjetoAtivo: IOTAProject;
var
  ProjectGroup: IOTAProjectGroup;
begin
  Result := nil;
  ProjectGroup := ProjectGroupAtivo;

  if Assigned(ProjectGroup) then
    if ProjectGroup.ProjectCount > 0 then
      Result := ProjectGroup.ActiveProject
end;

function ProjectGroupAtivo: IOTAProjectGroup;
var
  IModuleServices: IOTAModuleServices;
  IModule: IOTAModule;
  IProjectGroup: IOTAProjectGroup;
  i: Integer;
begin
  Result := nil;
  IModuleServices := BorlandIDEServices as IOTAModuleServices;
  for i := 0 to IModuleServices.ModuleCount - 1 do
  begin
    IModule := IModuleServices.Modules[i];
    if IModule.QueryInterface(IOTAProjectGroup, IProjectGroup) = S_OK then
    begin
      Result := IProjectGroup;
      Break;
    end;
  end;
end;

end.
```

## Nota do DevMan

Factory Method é padrão de projeto, basicamente um método de uma classe chamada Factory (fábrica) que cria e retorna um objeto de outra classe chamado Product, separando assim a lógica da criação de um objeto do seu uso. Ele permite também que as subclasses da fábrica definam qual o tipo correto de objeto a ser criado.

**Listagem 5.** Criador de arquivos de units

```
unit uCriadorArquivoUnit;

interface

uses
 Windows, SysUtils, ToolsApi;

 type
 TCriadorArquivoUnit =
class(TInterfacedObject,
  IOTAFile)
 private
  FDataCriacao: TDateTime;
 public
  constructor Create();
  function GetSource: string;
  function GetAge: TDateTime;
 end;

const

 CodigoUnit = [código da unit vazia vai aqui]

implementation

constructor TCriadorArquivoUnit.Create(g);
begin
 FDataCriacao := -1;
end;

function TCriadorArquivoUnit.GetAge: TDa-
teTime;
begin
 Result := FDataCriacao;
end;

function TCriadorArquivoUnit.GetSource:
string;
begin
 Result := CodigoUnit;
end;

end.
```

**Listagem 6**. Método GetOwner

```
function TCriadorUnit.GetOwner: IOTAModule;
begin
 Result := ProjectGroupAtivo;
 if Assigned(Result) then
  Result := (Result as IOTAProjectGroup).
  ActiveProject
 else
  Result := ProjetoAtivo;
end;
```

uma constante com uma string que é um esqueleto de uma classe básica (o código foi omitido por questões de espaço, mas contém basicamente o código de uma unit vazia). Aqui poderia também ser uma variável e essa string ser montada a partir de inputs do usuário (através de um form) ou até de informações vindas de um banco de dados.

Depois de implementado, o criador de arquivos pode ser passado ao criador de módulos. Um criador de módulos cria uma unit no projeto, mas usa o criador de arquivos para efetivamente criar a unit fisicamente com seu conteúdo. Para isso é necessário implementar as interfaces IOTACreator e IOTAModuleCreator.

Nos exemplos a seguir é criada uma nova unit chamada uCriadorUnit.pas com uma classe chamada TCriadorUnit que implementa IOTACreator e IOTAModuleCreator. Tanto a TCriadorUnit como a TCriadorArquivoUnit devem ser descendentes de TInterfacedObject. A uCriadorUnit deve referenciar a uCriadorArquivoUnit no seu uses.

Não é necessário implementar todos os métodos, então muitos deles ficam em branco ou com um retorno padrão, por exemplo string vazia se o tipo de retorno for string, false se o tipo de retorno for boolean e assim por diante, para que o Delphi decida automaticamente o melhor retorno em cada caso. Os métodos mais importantes para o exemplo:

- GetCreatorType;
- GetExisting;
- GetOwner;
- NewImplSource.

O primeiro método a ser implementado é o GetAncestorName. O código dele é simplesmente Result := ''; Isso porque o wizard desse exemplo não é um formulário e sim uma unit, e units não precisam de nome de ancestral.

O método GetAncestorType é análogo: retorna-se uma string vazia. O método GetCreatorType retornA sUnit, uma constante da ToolsAPI.pas cujo valor é a string 'Unit' e serve para que o modulecreator crie units sem formulários. O método GetExisting retorna false, assim o expert criará units novas sem nome, para que o desenvolvedor possa nomeá-las. O método GetOwner usa as funções criadas na unit uUtilidades para conseguir uma instância do ProjectGroup ou do Project ativo. Isso se faz necessário porque para criar uma unit ela deve ter um projeto que seja o owner dela. Esse método pode ser visto na **Listagem 6**.

Por último o método NewImplSource será o responsável por criar uma instância do TCriadorArquivoUnit. Esse método possui os parâmetros ModuleIdent, FormIdent e AncestorIdent. Esses parâmetros podem ser passados do NewImplSource de TCriadorUnit para o create de TCriadorArquivoUnit.

No exemplo deste artigo a maioria desses parâmetros serão ignorados. Eles servem para obter o nome da unit dado pelo Delphi e o nome do form e seu ancestral caso se trabalhe com formulários. A **Listagem 7** mostra como se dá a criação de um objeto do tipo TCriadorArquivoUnit e seu construtor Create não recebe parâmetro algum. Na **Listagem 8** pode-se ver o restante da classe TCriadorUnit.

Para transformar tudo isso em um wizard que crie arquivos é necessário criar uma classe que implemente as interfaces IOTAProjectWizard ou IOTAFormWizard, ou ambas. O método execute desta classe deve usar a variável pública global BorlandIDEServices na unit ToolsAPI para chamar o método CreateModule.

BorlandIDEServices implementa a interface IOTAModuleServices e faz parte de uma outra categoria de componentes da OTA que são os serviços. São classes que o próprio Delphi instancia e cujos métodos provêem funcionalidades úteis tanto para o próprio produto como para os desenvolvedores de wizards.

Ao se executar o método CreateModule deve-se passar a ele uma instância do criador de módulos:

```
(BorlandIDEServices as IOTAModuleServices).
CreateModule(Criador)
```

A **Listagem 9** mostra como é feito o wizard e onde ele entra em contato com o criador de módulos. Neste exemplo foram criados dois itens de menu para que um

**Listagem 7.** Método NewImplSource

```
function TCriadorUnit.NewImplSource(const
ModuleIdent, FormIdent,
 AncestorIdent: string): IOTAFile;
begin
 Result := TCriadorArquivoUnit.
 Create(ModuleIdent);
end;
```

menu seja colocado na raiz do menu do IDE e o outro seja submenu deste.

## Conclusão

Este artigo foi uma rápida pincelada no assunto OTA, que pode ser usada para diversas atividades, desde buscadores de palavras personalizados até code completion e code insight personalizados.

Espera-se que com esse conteúdo o leitor tenha uma introdução à criação de wizards e que possa assim criar tanto wizards para automatizar tarefas mecânicas do cotidiano como criar geradores de código para aumento de produtividade.

Também é possível criar validadores de código que podem avaliar o código da

**Listagem 8.** Classe TCriadorUnit

```
TCriadorUnit = class(TInterfacedObject, IOTACreator, IOTAModuleCreator)
 public
  //IOTACreator
  function GetCreatorType: string;
  function GetExisting: Boolean;
  function GetFileSystem: string;
  function GetOwner: IOTAModule;
  function GetUnnamed: Boolean;
  property CreatorType: string read GetCreatorType;
  property Existing: Boolean read GetExisting;
  property FileSystem: string read GetFileSystem;
  property Owner: IOTAModule read GetOwner;
  property Unnamed: Boolean read GetUnnamed;
  //IOTAModuleCreator
  function GetAncestorName: string;
  function GetImplFileName: string;
  function GetIntfFileName: string;
  function GetFormName: string;
  function GetMainForm: Boolean;
  function GetShowForm: Boolean;
  function GetShowSource: Boolean;
  function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
  function NewIntfSource(const ModuleIdent, FormIdent,
   AncestorIdent: string): IOTAFile;
  procedure FormCreated(const FormEditor: IOTAFormEditor);
  property AncestorName: string read GetAncestorName;
  property FormName: string read GetFormName;
  property ImplFileName: string read GetImplFileName;
  property IntfFileName: string read GetIntfFileName;
  property MainForm: Boolean read GetMainForm;
  property ShowForm: Boolean read GetShowForm;
  property ShowSource: Boolean read GetShowSource;
 end;

implementation

procedure TCriadorUnit.FormCreated(const FormEditor: IOTAFormEditor);
begin

end;

function TCriadorUnit.GetAncestorName: string;
begin
 Result := '';
end;

function TCriadorUnit.GetCreatorType: string;
begin
 Result := sUnit;
end;

function TCriadorUnit.GetExisting: Boolean;
begin
  Result := false;
end;

function TCriadorUnit.GetFileSystem: string;
begin
 Result := '';
end;

function TCriadorUnit.GetFormName: string;
begin
 Result := '';
end;

function TCriadorUnit.GetImplFileName: string;
begin
 Result := '';
end;

function TCriadorUnit.GetIntfFileName: string;
begin
 Result := '';
end;

function TCriadorUnit.GetMainForm: Boolean;
begin
 Result := false;
end;

function TCriadorUnit.GetShowForm: Boolean;
begin
 Result := false;
end;

function TCriadorUnit.GetShowSource: Boolean;
begin
 Result := true;
end;

function TCriadorUnit.GetUnnamed: Boolean;
begin
 Result := true;
end;

function TCriadorUnit.NewFormFile(const FormIdent,
 AncestorIdent: string): IOTAFile;
begin
 Result := nil;
end;

function TCriadorUnit.NewIntfSource(const ModuleIdent, FormIdent,
 AncestorIdent: string): IOTAFile;
begin
  Result := nil;
end;

end.
```

**Listagem 9.** Classe TWGeradorUnit

```
unit uWGeradorUnit;

interface

uses Dialogs, ToolsAPI, Menus;
type
 TWGeradorUnit = class(TNotifierObject, IOTAWizard, IOTARepositoryWizard,
   IOTAProjectWizard, IOTAFormWizard)
 private
  FRaizMenu: TMenuItem;
  FMenuItem: TMenuItem;
 public
  procedure MenuItemClick(Sender: TObject);
 public
  constructor Create;
  destructor Destroy; override;
  //iotawizard
  function GetIDString: string;
  function GetName: string;
  function GetState: TWizardState;
  procedure Execute;
  // IOTARepositoryWizard
  function GetAuthor: string;
  function GetComment: string;
  function GetPage: string;
  function GetGlyph: Cardinal;
 end;

procedure Register;

implementation

uses
 uCriadorUnit;

procedure TWGeradorUnit.MenuItemClick(Sender: TObject);
begin
 Execute;
end;

constructor TWGeradorUnit.Create;
begin
 inherited;
 FMenuItem := TMenuItem.Create(nil);
 FRaizMenu := TMenuItem.Create(nil);
 FRaizMenu.Caption := 'Experts Clube Delphi';
 FMenuItem.Caption := 'Criar Unit';
 FRaizMenu.Add(FMenuItem);
 FMenuItem.OnClick := MenuItemClick;
 (BorlandIDEServices as INTAServices).MainMenu.Items.Add(FRaizMenu);
end;

destructor TWGeradorUnit.Destroy;
begin
 FMenuItem.Free;
 FRaizMenu.Free;
 inherited;
end;

function TWGeradorUnit.GetIDString: string;
begin
 Result := 'Clube.Delphi.TWGeradorUnit';
end;

function TWGeradorUnit.GetName: string;
begin
 Result := 'GeradorUnit Expert'
end;

function TWGeradorUnit.GetState: TWizardState;
begin
 Result := [wsEnabled];
end;

procedure TWGeradorUnit.Execute;
var
 FUnit: IOTAModule;
begin
 FUnit := (BorlandIDEServices as
   IOTAModuleServices).CreateModule(TCriadorUnit.Create);
end;

function TWGeradorUnit.GetAuthor: string;
begin
 Result := 'Revista Clube Delphi';
end;

function TWGeradorUnit.GetComment: string;
begin
 Result := 'Gerador de esqueletos de classes';
end;

function TWGeradorUnit.GetGlyph: Cardinal;
begin
 Result := 0;
end;

function TWGeradorUnit.GetPage: string;
begin
 Result := 'Clube Delphi';
end;

procedure Register;
begin
 RegisterPackageWizard(TWGeradorUnit.Create);
end;
end.
```

equipe de programadores de acordo com certas métricas ou padronizações.

Os pacotes GEXperts e CNWizards são bons exemplos de ferramentas de produtividade que viabilizam certas tarefas, como alinhamento de controles visuais, em questão de segundos. São duas importantes fontes de consulta e referência, pois ambos já são usados pela comunidade Delphi desde a versão 3.

**Vitor Luiz Rubio**
*vitorrubio@gmail.com - http://www.vitorrubio.blogspot.com*
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Atua como Analista de Sistemas Sênior na Editora Revista dos Tribunais. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

**Dê seu feedback sobre esta edição!**

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

***www.devmedia.com.br/clubedelphi/feedback***

# CodeSite

## Log e depuração avançada no Delphi XE

Rodrigo Araujo

## Resumo DevMan

***De que trata o artigo:***

O artigo aborda os detalhes da utilização do Delphi XE com a ferramenta CodeSite, desde seus conceitos básicos, adição de mensagens para log de erros, até os exemplos concretos para depuração de códigos.

***Para que serve:***

Utilizando o Delphi XE, o objetivo principal deste artigo é motivar o uso da ferramenta CodeSite, servindo na análise e depuração de erros, com a intenção de corrigi-los.

***Em que situação o tema é útil:***

A depuração avançada é útil para encontrar falhas nos sistemas, que não são detectadas pelas formas convencionais de depuração, garantindo assim mais qualidade ao software final e satisfação do cliente.

***CodeSite:***

Este artigo aborda o CodeSite Express presente no Delphi XE, mostrando como adicionar mensagens, em busca de uma análise de erros de programação ou erros inesperados na execução de um aplicativo. Na introdução é caracterizado o que é o CodeSite, recursos básicos e a utilização de seus componentes principais. O CodeSite contém uma classe utilizada para enviar mensagens através de um método e uma outra classe que permite compartilhar propriedades entre suas instâncias. Possui um aplicativo responsável por encaminhar as mensagens ao seu destino, seja ele um visualizador ou um arquivo. O seu visualizador é representado pelo CodeSite Live Viewer que permite receber as mensagens no momento em que elas são enviadas pelo aplicativo responsável, ou de outra forma, recebendo um arquivo de log de mensagens previamente armazenado. Este arquivo poderá ser analisado no CodeSite Live Viewer com o objetivo de depurar o aplicativo e verificar o seu comportamento. Por fim serão demonstrados alguns exemplos básicos da utilização do CodeSite, seja para envio de mensagens simples, ou para a utilização de envio de mensagens através de Thread.

Ao executar um aplicativo, seja ele feito em Delphi XE ou outro ambiente qualquer, erros podem acontecer, sejam erros básicos no desenvolvimento do aplicativo ou erros de difícil análise. Dessa forma, o aplicativo poderá não se comportar da forma desejada e, às vezes, poderá ser extremamente difícil verificar e diagnosticar o que está realmente acontecendo de errado. Uma forma de detecção de erros é utilizar o método tradicional de depuração de códigos, através do debugger integrado do IDE. Mesmo assim poderá ser um trabalho demorado, e que talvez não seja obtido o resultado esperado, pois vários métodos podem estar relacionados sendo chamados em sequência, eventos podem ser acionados sem o controle total do desenvolvedor, e a chamada e utilização de várias rotinas e tarefas podem acontecer de forma simultânea.

O CodeSite, que está incluído em todas as versões do RAD Studio XE (Delphi XE, C++ Builder XE e Delphi Prism XE), auxilia com formas avançadas de depuração, através da adição de mensagens em determinados fluxos de códigos de uma aplicação, possibilitando verificar o comportamento do aplicativo. Ele fornece informações mais detalhadas de como o código está sendo executado, permitindo localizar problemas de uma maneira mais ágil. Assim, dará maior segurança para que o aplicativo seja executado corretamente, permitindo de uma forma eficaz a realização do ciclo de vida do aplicativo.

Possui algumas características principais como a não interrupção do fluxo de execução da aplicação, as suas classes de logs podem capturar vários tipos de informações, os dados de estruturas complexas são facilmente armazenados, possui fácil configuração e permite agrupar os logs em grupos definidos pelo usuário.

Além disso, o CodeSite realiza o tratamento de algumas deficiências que ocorrem com o tradicional método de depuração de aplicativos, por exemplo utilizando breakpoints, não sendo limitado a este modelo de depuração, permitindo ir mais além sobre o que e quando ficará registrado em um arquivo de log, dando maior flexibilidade no gerenciamento de suas informações para a análise dos logs gerados.

Para o seu funcionamento, existem três componentes principais utilizados na adição de logs em um aplicativo. Os componentes básicos do CodeSite são: CodeSite Logging Classes, CodeSite Dispatcher e CodeSite Viewers.

## CodeSite Logging Classes

É o componente que contém a classe responsável por possuir os métodos necessários para adicionar os logs de um aplicativo. Para o envio de mensagens, é utilizada a classe TCodeSiteLogger, onde o principal método é o Send, método este sobrecarregado, podendo enviar uma mensagem em formato de string simples ou outras formas derivadas incluindo envio de tipos de dados numéricos, datas, objetos e outros tipos, sem ter que realizar conversões de formatos, sendo este um grande benefício.

Cada mensagem gerada possui um tipo representado no CodeSite por um identificador. Existem dois tipos de mensagens: simples e inspeção. Uma mensagem simples é utilizada para destacar alguma informação e não necessita de nenhum identificador adicional associado ao seu conteúdo, simplesmente é realizada a chamada de uma mensagem com ou sem a passagem de parâmetros ou objetos. Já para uma mensagem de inspeção, deverá conter uma mensagem que permita verificar detalhes adicionais para realizar uma inspeção mais detalhada de seu conteúdo. Um exemplo deste tipo de mensagem é uma TException, que contém alguns detalhes adicionais indicando o provável motivo causador do erro.

Além disso, a classe TCodeSiteManager possui o objetivo de fornecer propriedades compartilhadas entre as instâncias ativas de TCodeSiteLogger. Dessa forma, podendo definir algumas propriedades em comum das instâncias, como por exemplo, DefaultDestination, que indicará o destino padrão das mensagens, podendo ser direcionadas para um visualizador, um arquivo ou ainda um endereço remoto (via TCP, UDP ou HTTP).

**Nota**

No Delphi XE, para um aplicativo utilizar as funcionalidades do CodeSite, basta incluir a unit CodeSiteLogging na cláusula uses.

**Nota**

O método Send é sobrecarregado para facilitar o envio de mensagens onde vários tipos de objetos podem ser utilizados sem a necessidade de realizar conversões de tipos.

## CodeSite Dispatcher

É o aplicativo responsável por encaminhar as mensagens enviadas e destinadas ao CodeSite, transportando elas ao seu destino final, seja um visualizador ou um arquivo de log, podendo estar em um endereço local ou em um local remoto. Possui algumas características, que podem ser customizadas, como definições de caminhos padrões para logs, definições de bloqueio de alguns tipos de mensagens e bloqueios por categorias.

O Dispatcher trata da sincronização de todas as mensagens enviadas, além de amenizar e facilitar o processo de envio de mensagens de um aplicativo, possibilitando manter o seu nível de desempenho, situação bastante cômoda para quem irá desenvolver, principalmente para aplicativos multitarefa (aplicações Web ou DataSnap por exemplo), onde o número de mensagens enviadas e a perfeita sincronização entre elas é a grande importância para um bom desempenho e uma correta interpretação das mensagens geradas para análise.

## CodeSite Viewers

O destino dado a uma mensagem para o CodeSite é realizado pelo CodeSite Dispatcher que irá transportar a mensagem ao seu destino final. Existem dois destinos básicos de destino para as mensagens: CodeSite Live Viewer e CodeSite Log File. Elas poderão estar localizadas na máquina local ou em uma máquina remota, além disso, existe a possibilidade de serem apresentadas em mais de um destino.

## CodeSite Live Viewer

É o visualizador padrão de mensagens do CodeSite. Sua visualização permite receber diretamente do CodeSite Dispatcher as mensagens online no momento em que elas são geradas, ou então, abrir um arquivo de log previamente armazenado.

Possibilita salvar os projetos e que sejam customizados de acordo com determinados critérios, podendo adicionar filtros para a sessão de depuração atual. Ao guardar essas informações, elas poderão ser facilmente restauradas em uma próxima vez em que ocorra necessidade de depurar a aplicação.

O seu layout permite visualizar de forma rápida e simples a inspeção das mensagens nele contidas. Sua janela principal é divida em painéis, onde o painel das mensagens (primeiro painel do lado superior esquerdo, Message Panel) é responsável por ser a área principal, local que receberá as mensagens, o segundo painel (o inferior esquerdo, Message Detail Panel) trará as informações associadas a determinada mensagem selecionada no painel Message Panel, o terceiro painel (primeiro do lado superior direito, Inspector Panel) mostrará as informações das propriedades e seus valores associados com a mensagem atualmente selecionada, e, por último, localizada no canto inferior direito está o Call Stack Panel responsável por indicar a pilha de chamadas de uma determinada mensagem.

A sua barra de ferramentas permite acesso a comandos que irão modificar o layout e a inspeção das mensagens. As mensagens recebidas poderão ser organizadas em views para uma melhor visualização. Em uma view é possível filtrar as mensagens e observar apenas o que realmente interessa, podendo organizar a view por mensagens recebidas de aplicativos ou por determinados tipos. Pode-se selecionar categorias de mensagens e visualizá-las na medida em que elas são recebidas, dessa forma adicionando e direcionando a mensagem em determinada categoria criada.

## CodeSite Log File

O envio de mensagens diretamente para um arquivo funciona de maneira semelhante a que ocorre com o CodeSite Live Viewer, diferenciando no sentido de que a mensagem é salva em arquivo físico no disco, que poderá ser analisada quantas vezes for necessário. Para salvar as mensagens em arquivo o CodeSite-Manager deverá receber na propriedade DefaultDestination o indicativo de que será encaminhado para um arquivo, sendo assim, o CodeSite Dispatcher saberá que o direcionamento das mensagens serão para este meio. A extensão de um arquivo utilizada é .csl (CodeSite Log) e contém todas as informações pertinentes às mensagens enviadas, podendo ser o arquivo analisado no CodeSite Live Viewer.

## CodeSite na prática

A **Listagem 1** apresenta algumas informações adicionadas ao Codesite através de um aplicativo de testes criado no Delphi XE, tais informações são: uma mensagem inicial ao Codesite indicando "Olá Codesite – Teste Inicial", "BtnTeste01Click executando..."com a passagem de parâmetro self, indicando as propriedades do formulário, "Propriedades do BtnTeste01" com a passagem de BtnTeste01 indicando as propriedades de um TButton, "Fonte do Formulário" contendo a fonte que o formulário está utilizando e um separador representado por Codesite.AddSeparator. A **Figura 1** representa a execução do aplicativo para visualizar o código apresentado.

No exemplo da **Figura 1** foram criadas seis mensagens e todas direcionadas ao CodeSite Live Viewer, possuindo quatro tipos de mensagens, a primeira sendo de informação, as três seguintes como sendo de objetos, a quinta mensagem de atenção e a última mensagem definindo uma linha como um separador.

A **Listagem 2** representa os códigos adicionados em um método chamado BtnCalcularClick, para apresentar as mensagens no Codesite, novamente, tendo como saída o CodeSite Live Viewer. Os códigos estão representados por: um EnterMethod, indicando a entrada no método em questão; mensagens para conferência dos valores de edits; a saída do método indicada por ExitMethod e um separador. O código em questão tem a finalidade de demonstrar a multiplicação de dois números levando em conta o código Try...Except para a adição de uma mensagem do tipo mensagem de inspeção, além de apresentar as fases do cálculo, os valores atribuídos a Edit1, Edit2 e o valor antes do cálculo de EdResultado e após realizar a multiplicação.

A **Figura 2** demonstra a execução do código e sua visualização no CodeSite Live Viewer. São realizados dois cálculos de multiplicação, no primeiro caso, uma multiplicação correta com valores inteiros repassados ao Edit1 e ao Edit2. Analisando no CodeSite Live Viewer é possível verificar suas propriedades, inclusive os valores informados ao Edit1, Edit2 e o resultado representado por EdResultado. No segundo caso um valor alfanumérico, representado pela letra "a" é atribuído, gerando uma exceção, que foi tratada no aplicativo, e representada pela mensagem em destaque permitindo, também, verificar a Call Stack de origem, no caso Início BtnCalcularClick. Dessa maneira, este exemplo, contém dois

**Listagem 1.** Código demonstrativo inicial de mensagens para o CodeSite

```
procedure TForm1.BtnTeste01Click(Sender: TObject);
begin
 CodeSite.Send('Olá CodeSite - Teste Inicial');
 CodeSite.Send('BtnTeste01Click executando...', self);
 CodeSite.Send('Propriedades do BtnTeste01' , BtnTeste01);
 CodeSite.Send('Fonte do Formulário', Form1.Font);
 CodeSite.AddSeparator;
end;
```

**Listagem 2**. Código demonstrativo ao informar valores a Edit1, Edit2 e pressionar o botão calcular

```
procedure TForm1.BtnCalcularClick(Sender: TObject);
begin
 CodeSite.EnterMethod(self, 'Inicio BntCalcularClick');
 CodeSite.Send('Realizando Cálculo');
 CodeSite.Send('Valor de Edit1', Edit1);
 CodeSite.Send('Valor de Edit2', Edit2);
 CodeSite.Send('Valor de EdResultado Antes Cálculo', EdResultado);
 // realizando o cálculo
 try
  EdResultado.Text := IntToStr(StrToInt( Edit1.Text) * StrToInt(Edit2.Text));
 except
  //em caso de erro gera exceção para codesite
  on e: exception do
   CodeSite.SendException(e);
 end;
 CodeSite.Send('Valor de EdResultado Após Cálculo', EdResultado);
 CodeSite.ExitMethod(self, 'Fim BtnCalcularClick');
 CodeSite.AddSeparator;
end;
```

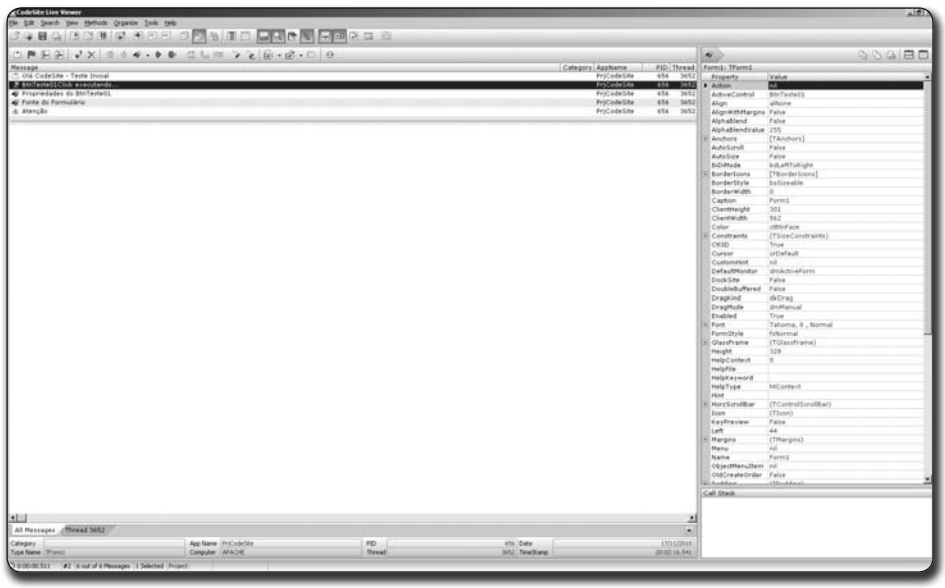


**Figura 1.** Execução do código da **Listagem 1**, a mensagem selecionada indica as propriedades do formulário

### Nota do DevMan

Call Stack, a pilha de execução de chamadas, contém a lista de todos os métodos chamados/executados até o momento do erro. Por ser uma pilha, os métodos executados logo antes do erro são posicionados no topo, e os métodos executados logo no início da execução do programa, que estão temporalmente distante do erro, estão no fim da pilha

**Listagem 3.** Código adicionado para o CodeSite na criação do formulário

```
procedure TForm1.FormCreate
  (Sender: TObject);
var
  saida : TCodeSiteDestination;
begin
  saida := TCodeSiteDestination.Create(self);
  saida.LogFile.Active := True;
  saida.LogFile.FilePath := 'c:\temp\Logs';
  saida.LogFile.FileName := 'Arquivolog.csl';
 //ativando salvará em arquivo e irá mostrar na tela
  saida.Viewer.Active := False;
  //saida.LogFile.MaxSize := 50;
  //tamanho máximo do arquivo
  //saida.LogFile.MaxParts := 10;
  // número máximo de arquivos
  CodeSiteManager.DefaultDestination := saida;
end;
```

tipos de mensagens: as mensagens simples e uma mensagem de inspeção, além de conter a entrada no método representada por Início BtnCalcularClick e a saída do método descrita por Fim BtnCalcularClick.

Utilizando o CodeSiteManager para alterar o destino padrão das mensagens, o CodeSite poderá ser utilizado para armazenar em arquivos de logs as execuções de códigos do Delphi XE. Assim, ao enviar uma mensagem, o CodeSite Dispatcher saberá que deverá salvar em arquivo a mensagem.

A **Listagem 3** demonstra o código adicionado ao FormCreate para dar suporte ao armazenamento em disco das mensagens geradas. Existe a indicação de que o LogFile estará ativo, além da configuração do FilePath e FileName contendo a extensão .csl (CodeSite Log), e onde este arquivo é visualizado no CodeSite File Viewer.

Na **Figura 3** é visualizado o arquivo de log gerado pela execução do aplicativo, ao abrir o arquivo ele é analisado no CodeSite File Viewer. Vale lembrar que o CodeSite File Viewer somente será executado quando o arquivo for aberto, pois na definição do código ele está indicado como Viewer. Active := False. Alterando a propriedade para True, as mensagens serão armazenadas em arquivo de log e também visualizadas em tempo de execução no CodeSite File Viewer.

As opções MaxSize e MaxParts permitem salvar os arquivos de log utilizando parâmetros opcionais, desta forma o tamanho máximo de um arquivo é definido em MaxSize e o número máximo de partes é definido em MaxParts. Os nomes de arquivos seguiriam o exemplo: Arquivolog_01.csl, Arquivolog_02.csl, Arquivolog_03.csl e assim por diante.

O próximo exemplo demonstra a utilização do CodeSite com Threads. Para melhor exemplificar, vamos considerar uma "corrida entre um leão e um leopardo". Cada um deles será representado por uma Thre-

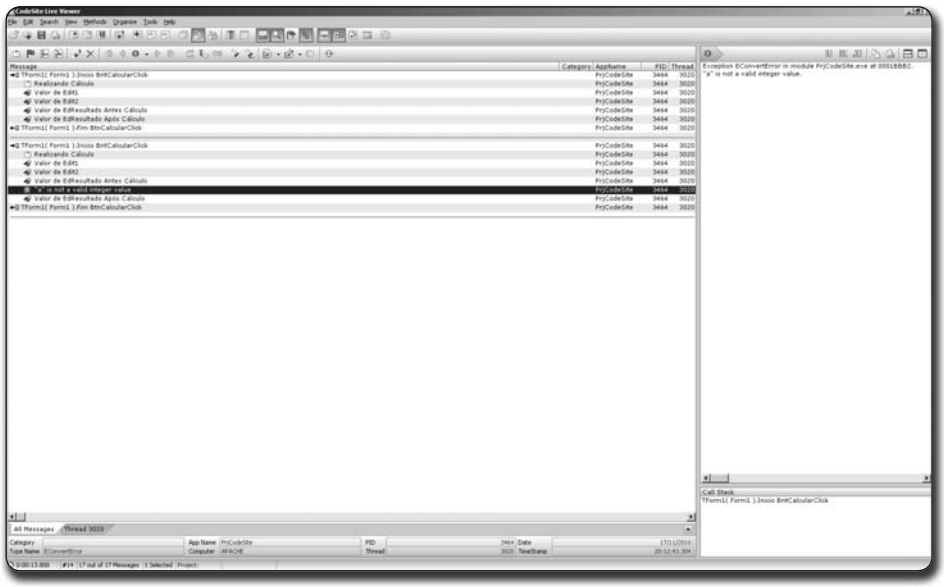


**Figura 2.** Mensagens enviadas ao realizar um cálculo de multiplicação

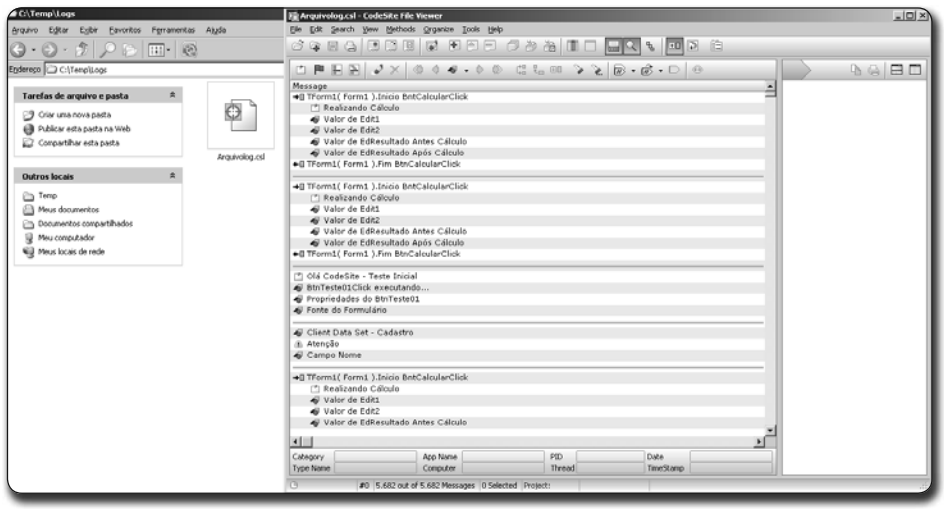


**Figura 3.** Arquivo de logs salvo na pasta C:\temp\logs (ao lado esquerdo da imagem) e o CodeSite Live Viewer com o arquivo aberto (ao lado direito)

**Nota**

Para os grupos de mensagens simples e inspeção existem outros tipos que podem ser utilizados: mensagens de notas, erros, checkpoints, formatos customizados, string lists, collections, enter methods e exit methods.

**Nota**

A propriedade Viewer.Active, se definia como True, possibilita a utilização do CodeSite em mais de um dispositivo de saída, no caso um arquivo e diretamente no CodeSite Live Viewer.

**Nota**

Uma View é descrita como uma nova aba de visualização definida para cada Thread utilizada no exemplo.

**Nota**

De acordo com a definição das procedures, a variável de nome "passada" indica que o leopardo percorre uma distância maior por passada que o leão e realizando um simples cálculo matemático, o leopardo é mais rápido.

ad com características próprias relativas à sua passada de corrida, onde percorrem uma mesma distância pré-definida. O objetivo é realizar uma corrida entre os dois animais, onde a corrida será representada pelo somatório da distância das suas passadas até atingirem a distância definida. A **Listagem 4** indica a definição das Threads para o leão e para o leopardo.

O código das **Listagens 5** e **6** descreve as implementações de código para o leão e leopardo e a adição de mensagens ao CodeSite utilizando o modelo de Threads. No código é adicionado um nome para a Thread descrito por SetCurrentThreadName, a entrada no método, uma mensagem a cada iteração do código (passada de corrida) e, ao finalizar, indica que terminou a corrida.

Na **Listagem 7** são visualizados os códigos dos botões que iniciam a corrida do leão e do leopardo. A execução do aplicativo demonstrado na **Figura 4** indica o Codesite File Viewer com as mensagens adicionadas ao visualizador. A **Figura 5** informa as iterações realizadas por cada um dos animais (Thread) e indica quem foi o vencedor da corrida.

No CodeSite Live Viewer é possível verificar o vencedor da corrida, na parte inferior de cada View existe o número de mensagens enviadas, no caso do leão foram 52 mensagens e para o leopardo 35,

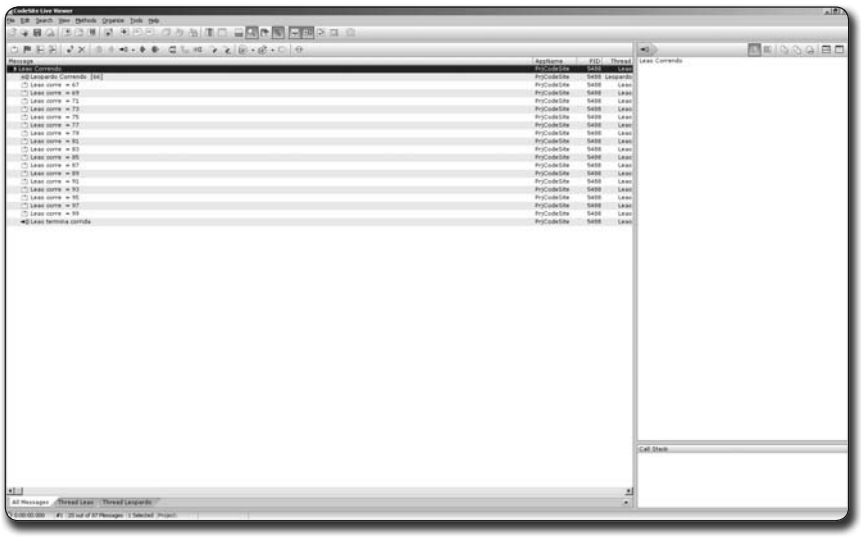


**Figura 4.** Descrição da corrida no CodeSite Live Viewer indicando os processos diferenciados por Thread (leão e leopardo)

**Listagem 4.** Definição das Threads de leão e leopardo

```
TLeao=class(TThread)
protected
  procedure Execute; override;
end;

TLeopardo=class(TThread)
protected
  procedure Execute; override;
end;
```

**Listagem 5**. Criação da procedure para o leão e adição de mensagens ao CodeSite

```
procedure TLeao.Execute;
var
  contador, passada : integer;
begin
  CodeSiteManager.SetCurrentThreadName('Leao');
  CodeSite.EnterMethod('Leao Correndo');
  passada := 2; //leao corre 2m/s
  contador := 1;
  while contador < 100 do //irá correr 100m
  begin
    CodeSite.Send('Leao corre ', contador);
    sleep(5);
    contador := contador + passada;
  end;
  inherited;
  CodeSite.EnterMethod('Leao termina corrida');
end;
```

**Listagem 6**. Criação da procedure para o leopardo e adição de mensagens ao CodeSite

```
procedure TLeopardo.Execute;
var
  contador, passada : integer;
begin
  CodeSiteManager.SetCurrentThreadName('Leopardo');
  CodeSite.EnterMethod('Leopardo Correndo');
  passada := 3;  //leopardo corre 3m/s
  contador := 1;
  while contador < 100 do //irá correr 100m
  begin
    CodeSite.Send('Leopardo corre ', contador);
    sleep(5);
    contador := contador + passada;
  end;
  inherited;
  CodeSite.ExitMethod('Leopardo termina corrida');
end;
```

## Nota do DevMan

**Threads e Processos**

Quando um aplicativos está em memória, utilizando recursos do processador, ele também é conhecido como processo. O Windows realiza multiprocessamento quando gerencia vários processos. Um processo na verdade não executa código, ele contém um objeto interno que é responsável pela execução dos códigos. Este objeto interno é chamado de Thread. Cada processo possui pelo menos um Thread, chamado de Thread primário ou principal. Um aplicativo pode fazer multiprocessamento através da manipulação de vários Threads. Uma aplicação multithread é aquela capaz de realizar multiprocessamento dentro de si própria. Os processos e Threads no Windows recebem um identificador quando executados (chamados respectivamente de ThreadID – identificador de Thread, e ProcessID – identificador de processo (PID)). Threads e Processos possuem prioridades de execução, como: baixa, normal ou alta, que são utilizadas pelo Windows quando ele precisa fornecer fatias de tempo (chamadas quantums) para os Threads de cada processo que está na memória. No Delphi, a classe TThread é utilizada para criar aplicações multithread.

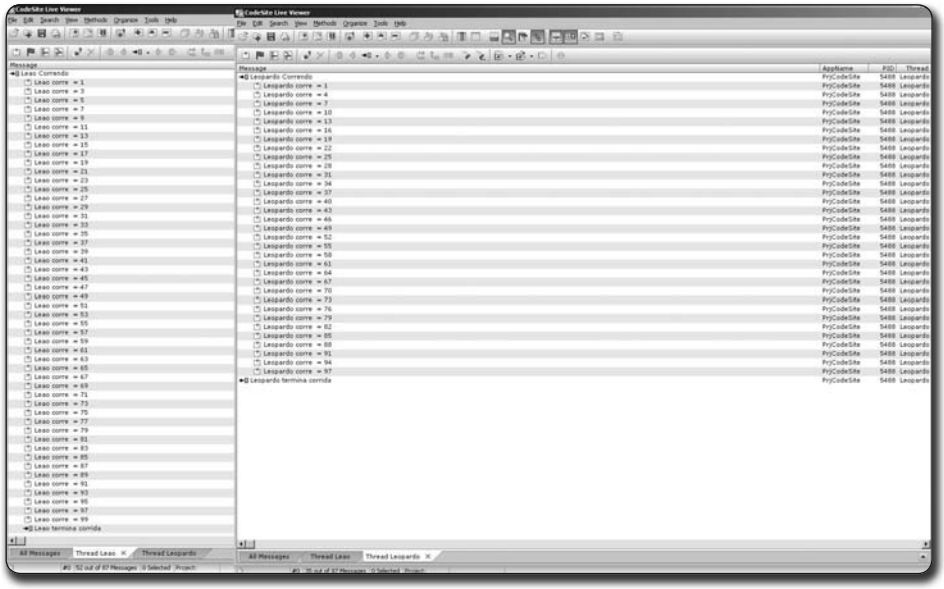


**Figura 5.** Lado a lado a Thread do leão e a Thread do leopardo

**Listagem 7**. BtnCorridaClick indicando o código que dará a largada da corrida

```
procedure TForm1.BtnCorridaClick(Sender:
TObject);
var
  leao : TLeao;
  leopardo :TLeopardo;
begin
  leao := TLeao.Create;
  leao.FreeOnTerminate := true;
  leao.Resume;
  leopardo := TLeopardo.Create;
  leopardo.FreeOnTerminate := true;
  leopardo.Resume;
end;
```

dessa forma, o leopardo atingiu primeiro a linha de chegada pré-determinada vencendo a corrida.

## Conclusão

No desenvolvimento de aplicações utilizando o Delphi XE, uma simples e excelente ferramenta para auxiliar na depuração de erros em aplicações é o CodeSite. Ela pode ser utilizada para depurar uma aplicação utilizando seu CodeSite Live Viewer ou armazenando as informações de logs em arquivos para posterior análise. Além disso, as mensagens podem ser enviadas para mais de um destino o que pode facilitar a interpretação e acelerar a correção do aplicativo. Ao decorrer do artigo, alguns assuntos relacionados à definição do CodeSite, suas características e principais funcionalidades foram tratadas, possibilitando que os conceitos básicos de sua estrutura fossem demonstrados. Por final, alguns exemplos de utilização da ferramenta foram criados com o objetivo de melhor transparecer o uso do CodeSite, utilizando mensagens simples e mensagens de inspeção para que seus conceitos fossem completamente tratados. Vale lembrar que algumas funcionalidades multitarefa com a execução de mais de uma Thread possibilitaram demonstrar o potencial do CodeSite na depuração de um aplicativo utilizando Delphi XE.

**Rodrigo Araujo**
*rodrigo.arsi@gmail.com*
Bacharel em Ciência da Computação pela UFSM – Santa Maria/RS trabalha com Análise e Desenvolvimento de Software há mais de 10 anos. Desde 2003, utiliza tecnologias .NET.

**CodeSite**
*http://www.raize.com/devtools/codesite/*

**Dê seu feedback sobre esta edição!**

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

***www.devmedia.com.br/clubedelphi/feedback***

# Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

**Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8000 artigos, além de palestras e vídeos com formações completas.**

TODAS as vídeo aulas que você sempre quis ver a partir de

**R$ 0,99**

**ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!**

**Saiba mais sobre o Sistema de Créditos!**

**www.devmedia.com.br/creditos**

DevMedia

# Desenvolvimento ágil

## Entenda a relação entre desenvolvimento ágil e

Antes de tudo é preciso saber que não importa se o teste será feito com técnicas automatizadas ou não, o importante e testar. Existem outras maneiras de testar o software, como por exemplo, contratar pessoas especialmente para esta função. Esta provavelmente não é a melhor alternativa, mas é possível e deve trazer bons resultados. O importante é executar os testes não importa se eles serão automatizados ou não.

Por outro lado, além de tornar o software mais barato, quando você automatiza os testes você também codifica mais rápido e quando tiver que fazer modificações poderá fazê-las mais facilmente e com mais segurança: você não estará preocupado se sua última alteração estragou outras partes do sistema.

É comum que as pessoas se assustem quando começam a estudar sobre teste unitário, pois existem muitos termos técnicos sobre o assunto como "caixa branca", "testes de regressão", "fases de testes", entre outras. Isso é o suficiente para assustar alguém que provavelmente não vai mais nem querer ouvir falar em técnicas de testes. Todos estes termos fazem os testes parecerem complicados, mas não são. Vou explicar o porquê.

### Por que os testes são simples

Existe uma sequência natural para o processo de desenvolvimento de um projeto e os testes estão nesta seqüência. Quando você começa a criar um programa ou uma nova parte dele, naturalmente a primeira coisa que você precisa é saber o que esta nova função deve fazer, sendo assim os testes devem ser criados para validar se a nova opção faz o que deveria fazer.


### Resumo DevMan

***De que trata o artigo:***

Este artigo explica o que é desenvolvimento ágil e ensina como criar um projeto seguindo o princípio de desenvolvimento dirigido por testes. Até o final do artigo você aprenderá como executar testes unitários com DUnit e como usar o Discover para saber quanto do seu código ainda não foi testado, além de entender quais são as técnicas mais interessantes para testar um software.

***Para que serve:***

O desenvolvimento dirigido por testes ajuda a codificar mais rápido, programas que terão menos falhas. Um programa que foi desenvolvido com metodologias ágeis normalmente são mais claros e flexíveis diante das mudanças do dia-a-dia.

***Em que situação o tema é útil:***

Executar testes é uma tarefa imprescindível para o desenvolvimento de um bom software. Quando eles são aplicados juntamente com o uso da ferramenta Discover, você pode chegar a ter 100% de todas as linhas do seu código testadas. Isto com certeza fará seu projeto amadurecer.

***Desenvolvimento Ágil:***

Os testes automatizados ajudam manter o projeto estável mesmo após grandes mudanças. Este artigo explica o que são testes unitários e que eles têm a ver com programação ágil além de ensinar como criar testes automatizados com o DUnit, que vem integrado ao Delphi desde a versão 2005. No final de tudo, você aprenderá como usar o Discover para saber se os testes deixaram escapar algum bug.


Depois que os testes passarem o sistema estará fazendo o que precisa: a próxima etapa é garantir que ele faz do jeito certo. Para isto outros testes serão executados, seja por você mesmo ou por outras pessoas. Nesta etapa serão testados itens como desempenho em situações extremas, facilidade de uso, clareza das mensagens, padronização das telas, funcionalidade entre outras coisas: o objetivo é refinar a opção. Existe um tipo de teste que se adapta melhor para cada tipo de "refinamento", por este motivo existem tantos termos técnicos. Não se concentre de mais com o nome de cada fase (refinamento): tenha em mente que cada opção deve passar por várias fases de testes para melhorar a qualidade.

# testes unitários

Rafael Stavarengo

Todo este processo é natural. Primeiro o sistema é codificado par afazer o que precisa fazer. Depois uma série de procedimentos são executados para refinar o programa e garantir que ele faz o que deve fazer do jeito certo. Você poderia fazer isso sem estudar nada sobre testes, mas seria mais difícil. As técnicas de testes existem para facilitar o trabalho de refinamento e aumentar a qualidade do software.

## Teste de software: o que é?

Teste de software é qualquer procedimento com o objetivo de provar que o software não funciona corretamente. Estes procedimentos podem ser executados por uma pessoa ou podem ser automatizados em pilhas de testes que ficaram cada vez mais completas à medida que o projeto madurece.

A diferença mais evidente entre um projeto desenvolvido orientado a testes e outro não orientado a testes está no processo de desenvolvimento. Enquanto que um começa a partir do teste automatizado e só é criado o necessário para o teste passar, o outro é simplesmente construído e em algum momento no futuro os testes serão executados por um humano. Por serem executados por humanos não há garantia de que os testes serão fielmente repetidos sempre que algo mudar.

É comum pensar que criar classes de testes aumenta o tempo de desenvolvimento do software tornando-o mais caro e até mesmo inviável. Este é um pensamento equivocado, pois de modo geral programas desenvolvidos com base em testes são mais simples e fazem somente o que devem fazer uma vez que será escrito somente o necessário para o teste passar. Dificilmente o tempo é desperdiçado com algo que não seja realmente necessário.

Quem programa com bases nos testes que cria, ganha tempo quando precisa encontrar um erro, pois raramente será necessário debugar o programa para encontrar um erro: basta rodar o teste e ele aponta onde exatamente está o problema. Algumas metodologias de desenvolvimento de software nos levam a criar classes de testes antes mesmo de criar a opção em si: este a meu ver é o cenário ideal. No decorrer deste artigo você verá como fazer isso.

Escrever testes para todas as classes que você criar deve ser uma realidade no seu dia-a-dia. Além de ser um requisito do modelo ISO de qualidade de software, os testes levarão o seu projeto para outro nível de qualidade. É claro que a qualidade dos testes, e consequentemente do produto final, depende diretamente da pessoa envolvida, mas mesmo assim um programa desenvolvido orientado a testes normalmente consegue atingir o objetivo com mais clareza e em menos tempo.

## Teste de software: o que NÃO é?

Testar um software não é debugar. Ao contrário da fase de codificação, cujo objetivo é construir o software, o teste é um processo que visa desmontá-lo para provar que ele não funciona ou que tem erros. Os testes devem ser escritos como se quem os escreve tivesse sido desafiado a provar que o programa tem falhas, como se já soubesse que as falhas existem e encontrá-las é apenas uma questão de tempo.

É difícil garantir que um software não tem erros, visto que normalmente existem muitas variáveis para avaliar, o que aumenta a complexidade. Um software pode falhar por vários motivos, como: problemas de hardware, erros de codificação, requisitos errados, mensagens pouco claras, entre outros motivos. A quantidade de pessoas envolvidas no projeto e qualificação de cada uma também influencia diretamente na qualidade do produto final. Além disso tudo, ainda tem o fato de que as alterações são constantes e normalmente muitas ao mesmo tempo: parece um ambiente caótico para quem não conhece.

Ainda hoje existem pessoas que pensam nos testes como uma tarefa secundária, porém há muito tempo escrever testes passou a ser uma fase do desenvolvimento assim como a análise e implantação. Os testes fornecem informações preciosas sobre o nível de qualidade de software em relação ao contexto que ele pertence.

## Modelo de qualidade

Existe um modelo base de qualidade, ou seja, existem documentos que descrevem como um programa deve se comportar para ser considerado de boa qualidade. O modelo ISO de qualidade de software considera quatro categorias diferentes para as falhas de um software, são elas: processo de desenvolvimento, qualidade interna, qualidade externa e qualidade percebida pelo usuário. Estas categorias englobam pelo menos seis características que não podem faltar em um software: usabilidade, funcionalidade, confiabilidade, eficiência, portabilidade e manutenibilidade.

Alcançar os níveis mais elevados de qualidade definidos pelos órgãos responsáveis não é uma tarefa muito simples. Para facilitar este trabalho os testes foram divididos em fases diferentes, cada uma delas focada em testar um determinado aspecto do sistema. É sobre estas fases que você verá a seguir.

**Listagem 1**. Exemplo de uma classe de teste caixa branca

```
01  unit TestPedidoDao;
02  interface
03  uses
04   TestFramework, PedidoDao, Generics.Collections, Pedido;
05  type
06   TestTPedidoDao = class(TTestCase)
07   strict private
08    FPedidoDao: TPedidoDao;
09   public
10    procedure SetUp; override;
11    procedure TearDown; override;
12   published
13    procedure TestPost;
14   end;
15  implementation
16  procedure TestTPedidoDao.SetUp;
17  begin
18   FPedidoDao := TPedidoDao.Create;
19  end;
20  procedure TestTPedidoDao.TearDown;
21  begin
22   FPedidoDao.Free;
23   FPedidoDao := nil;
24  end;
25  procedure TestTPedidoDao.TestPost;
26  var
27   Pedido: TPedido;
28  begin
29   CheckEquals(0, FPedidoDao.Count, 'A base deve estar vazia');
30   Pedido := TPedido.Create;
31   Pedido.Id := 1;
32   FPedidoDao.Post(Pedido);
33   CheckEquals(1, FPedidoDao.Count, 'Não salvou o primeiro pedido');
34   FPedidoDao.Post(Pedido);
35   CheckEquals(1, FPedidoDao.Count, 'Adicionou duas vezes o mesmo pedido');
36   Pedido := TPedido.Create;
37   Pedido.Id := 2;
38   FPedidoDao.Post(Pedido);
39   CheckEquals(2, FPedidoDao.Count, 'Não adicinou o segundo pedido');
40  end;
41  initialization
42   RegisterTest(TestTPedidoDao.Suite);
43  end.
```

## Fases de teste

Se você testa o seu projeto, então provavelmente já executa testes para a maioria das fases que cito a seguir, mesmo sem saber que elas existem. Isto acontece porque o processo de teste é algo natural e você não precisa conhecer as fases para poder executá-lo, embora conhecê-las facilita o trabalho e aumenta o profissionalismo.

É a fase de teste que o projeto se encontra que diz qual o objetivo do teste que deve ser executado e a partir daí é possível saber qual técnica de teste é a mais apropriada para o "tal objetivo". A seguir eu cito algumas fases de testes e posteriormente as técnicas mais interessantes.

Existe uma fase chamada "teste de unidade". Ela acontece ao mesmo tempo em que as opções estão sendo criadas. O objetivo é testar unidade por unidade do programa: cada classe/arquivo deve ter pelo menos outra classe de teste especialmente para ela. As técnicas de caixa branca e caixa preta são excelentes exemplos de técnicas para esta fase, pois para testar as unidades o testador precisa necessariamente ter acesso ao código fonte do projeto. Equipes que trabalham com desenvolvimento ágil normalmente criaram estes testes antes de começar a codificar a opção propriamente dita.

Outra fase é a "teste de integração". Nesta fase o objetivo é garantir que a comunicação esteja fluindo bem entre os componentes (ou módulos, como preferir), do sistema. Mais uma vez deve existir uma série de classes de testes para garantir, por exemplo, que uma venda não será feita em quanto um balanço estiver em execução. Este teste também pode ser executado por um humano interagindo com a interface do programa, embora também existam maneiras de automatizar testes que interagem com a interface como se fossem usuários.

Os testes da categoria não funcionais são executados nas fases "Testes de desempenho" e "testes de carga". Como o próprio nome diz, nestas fases o objetivo é descobrir qual o limite de informação que o sistema consegue processar até que seu desempenho seja considerado inaceitável. Depois desta fase será mais fácil identificar quais os requisitos mínimos para o software.

Existem ainda outras fases como, por exemplo, "testes de sistema" e "testes de aceitação". Durante a fase "testes de sistema" os testes são executados por alguém que realmente entende como o usuário irá trabalhar no dia-a-dia, pois nesta fase o sistema deve ser usado como de fato será usado quando estiver em produção. Posteriormente vem a fase dos testes de aceitação. Estes testes devem ser executados por um grupo seleto de usuário que dirão se o programa é aceitável para a realidade.

## Técnicas de testes

Você viu que para facilitar o trabalho de testar um projeto foram criadas fases de testes cada uma com o objetivo de descobrir falhas de categorias diferentes. Além das fases, também existem técnicas de testes. Algumas das técnicas mais populares são: caixa branca, caixa preta, e regressão. Cada uma destas técnicas funciona melhor para encontrar falhas de uma ou mais categorias.

Enquanto a fase do teste diz qual o tipo de falha deve ser procurada, as técnicas ajudam a encontrar as falhas propriamente ditas. Por exemplo, durante a fase de teste de desempenho o objetivo é encontrar falhas que ocorrem quando o software estiver no limite de processamento e a técnica de testes não funcionais é a melhor para encontrar falhas desta categoria.

As técnicas e fases existem para facilitar o trabalho de testar um software sem deixar de testar nenhuma característica. É como se todas estas fases e técnicas juntas formassem um protocolo de testes, que quando seguido garantem um "selo de qualidade superior".

A seguir você verá um pouco sobre as técnicas de testes mais populares. Estas técnicas podem ser usadas para encontrar falhas de mais de uma categoria.

## Caixa Branca

Testes de caixa branca são criados pela pessoa que tem acesso ao código fonte, por isso fica a cargo do próprio programador que criou a opção. Este é um tipo de teste bem difundido entre os desenvolvedores. Neste teste o programador deve criar uma classe de teste para garantir que o comportamento interno de outra classe esteja correto. O objetivo é garantir que a classe testada faça o que deve fazer sem erros.

O teste de caixa branca também é conhecido como testes orientados a lógica, já que o programador precisa conhecer como a classe a ser testada funciona internamente. O resultado deve ser um teste que valide todos os caminhos possíveis – todos os "if's" e "cases" opcionais – é com esta técnica de teste que o programador garante o bom fluxo de informações da classe testada.

Frameworks como JUnit e DUnit, entre outros, são exemplos de ferramentas para criar testes do tipo caixa branca. Estes frameworks facilitam a vida do programador, pois a maioria do "trabalho sujo" necessário para criar classes de testes já está pronto, restando para o programador simplesmente criar os métodos de testes. O DUnit vem integrado ao Delphi desde a versão 2005. Mais adiante voltarei a falar sobre ele. Por hora veja um exemplo de teste caixa branca feito com DUnit, na **Listagem 1**.

## Caixa Preta

O teste caixa preta, também conhecido como teste funcional é o complemento do teste caixa branca e vice-versa. Enquanto o teste caixa branca concentra-se no comportamento interno da classe, o teste caixa preta concentra-se no comportamento externo. Eles são usados para demonstrar que as funções do software são operacionais.

De modo grosseiro, para executar este tipo de teste deve-se levantar todas as entradas possíveis para então passá-las para o software que vai processar e devolver o resultado que será comparado com o valor que se espera para cada parâmetro de entrada. As entradas variam de acordo com a estrutura do programa e podem ser infinitas dependendo do contexto: imagine quantas entradas diferentes existem para um programa que aceita um ponto flutuante como parâmetro. O ideal é sempre testar todas as entradas existentes, mas quando isto se torna inviável o testador deve escolher um conjunto de entradas que garanta um teste mais completo possível.

O testador – que não precisa ser necessariamente o programador – deve garantir que a saída de dados está de acordo com o esperado para cada entrada. Estes testes geralmente identificam erros de categorias bem específicas, como: erro de inicialização, erro de interface, erro de conexão com o banco de dados, erro de estrutura de dados, entre outros. Repare que todos estes erros são referentes ao comportamento externo do programa, onde a técnica caixa preta atua.

## Testes Caixa Cinza e não funcionais

Os testes de caixa branca e caixa preta complementam um ao outro, pois sozinhos não são capazes de identificar todos os erros e riscos que envolvem um programa. O teste caixa cinza é a junção destes dois testes.

A técnica de testes não funcionais visa garantir que o software funciona mesmo em situações extremas. São os testes não funcionais que garantem o bom comportamento do sistema quando, por exemplo, a conexão com o banco de dados cair inesperadamente. Para executar os testes não funcionais também é comum usar uma grande quantidade de dados para descobrir qual o limite do sistema. Após passar por uma boa série de testes não funcionais o software provavelmente será capaz de manter o controle da situação e continuar funcionando mesmo quando algo inesperado sair errado.

## Testes de Regressão

Normalmente você aplica todos os testes para cada nova opção ou alteração do projeto e quando chegar ao final de uma versão, todos os testes devem ser executados novamente para garantir que nenhuma outra parte parou de funcionar por conta das modificações feitas durante o desenvolvimento da versão. Executar todos os testes criados desde a primeira versão é uma técnica chamada de "teste de regressão". Quando se cria testes automatizados você tem a opção de executá-los todos de uma vez a qualquer momento, para garantir que as alterações não influenciaram o que estava funcionando: esta é mais uma (enorme), vantagem dos testes unitários. Imagine alterar grandes pedaços do projeto e ter certeza de que as outras partes não foram danificadas – testes unitários tornam o software mais maduro e deixam o programador mais seguro do que faz.

"Teste de regressão" significa que todos os testes já criados desde a primeira versão do projeto serão executados novamente para garantir que tudo saiu como esperado – este é um teste necessário para garantir que as modificações não causaram novos erros. Esta técnica normalmente é usada no final de cada ciclo de desenvolvimento.

## Metodologias de Desenvolvimento Ágil

Algo comum hoje em dia é encontrar vagas para desenvolvedores cujo um dos requisitos é ter participado de alguma equipe que trabalhe com desenvolvimento ágil. Metodologia ágil é um conceito que visa trazer uma série de vantagens para a equipe diminuindo os riscos que envolvem produzir um software e tem como característica a velocidade com que novas funções são implementadas quando comparada com a metodologia convencional: semanas ao invés de meses.

Uma equipe de desenvolvimento ágil normalmente tem comunicação em tempo real tanto entre os integrantes da equipe quanto entre a equipe e as pessoas que conhecem a definição do programa – por exemplo, os clientes. Para tornar o processo mais ágil esta comunicação deve ser prefe-

### Nota do DevMan

O gerenciamento de informações de um cadastro – salvar, buscar, atualizar, excluir – também é conhecido como CRUD. CRUD é um acrônimo da uma expressão inglesa, cujas letras significam: Create (salvar), Retrieve (buscar), Update (atualizar) e Delete (excluir).

rencialmente feita frente a frente ao invés de usar documentos – talvez por isso a baixa documentação seja considerada uma desvantagem das metodologias ágeis.

Um dos princípios dos métodos ágeis é que toda alteração deve trazer uma nova função e para conseguir isto o desenvolvimento é divido em pequenos ciclos, como se cada ciclo fosse um pequeno projeto dentro do projeto verdadeiro – um novo ciclo é iniciado sempre que alguma função nova precisar ser implementada ou uma antiga precise ser alterada. Estes ciclos envolvem passos pré-determinados, que mudam de acordo com a metodologia.

Existem diversos métodos de desenvolvimento ágil e o órgão responsável por promover estes métodos é o The Agile Alliance. Segundo o site do The Agile Alliance, eles promovem os princípios de desenvolvimento ágil porque acreditam que estas metodologias tornam o software mais produtivo humano e sustentável. Uma das coisas que este órgão faz é manter o The Agile Manifesto (O Manifesto Ágil), onde está descrito quais são os valores e princípios do desenvolvimento ágil. Na sessão Links do artigo está um link para o site do The Agile Alliance, onde existe, entre outras coisas, o The Agile Manifesto.

Como disse, existe mais de um método de desenvolvimento ágil, como por exemplo: Scrum, Dynamic System Development Method, Feature Driven Development, Extreme Programing e Test Driven Development, entre outros. A seguir vou falar um pouco sobre o método Test Driven Development e depois mostrarei o DUnit como uma alternativa para criar testes automatizados.

## TDD – Test Driven Development

Test Driven Development que se traduz como Desenvolvimento Orientado a Testes ou simplesmente TDD, assim como qualquer metodologia de desenvolvimento ágil é composta por pequenos ciclos de desenvolvimentos. Estes ciclos são formados por cinco passos: 1º) Criar um teste; 2º) Executar o novo teste; 3º) Criar a nova função; 4º) Executar os testes novamente; 5º) Refatorar o código.

A primeira coisa que deve ser feita para começar alterar ou criar qualquer opção dentro de um software seguindo o método orientado por testes, é obviamente criar um teste. Este primeiro teste deve ser criado antes da opção em si, por este motivo quando o teste for executado a primeira vez ele falha obrigatoriamente, já que a opção ainda não existe. Depois disto a opção começa a ser desenvolvida com o objetivo de fazer os testes passarem. Quando tudo estiver certo e o teste não falhar mais, chega a hora de refatorar o código para tornar o projeto mais flexível e reutilizável – depois da refatoração você pode executar os testes novamente para garantir que a opção desenvolvida não parou de funcionar.

Este ciclo não pode ser muito demorado – não deve passar de quatro semanas e o ideal seria uma semana. Se o ciclo demora muito para terminar os testes dos outros integrantes da equipe começam a falhar por conta da opção que ainda não está concluída. Este método de desenvolvimento é focado nos requisitos do software, pois o programador começa escrevendo testes validando somente o que pede o requisito. Consequentemente a opção desenvolvida será bem enxuta e objetiva, já que o programador escreve apenas o código necessário para fazer os testes passarem. Sem perceber o programador acaba seguindo princípios como KISS (Keep It Simple Stupid) e You Ain't Gonna Need It (não adicionar funcionalidade até que seja necessário).

## DUnit

Desde a versão 2005 o Delphi traz integrado ao IDE o DUnit. Este framework implementa tudo o que você precisa para começar a escrever classes de testes unitários sem precisar configurar nada, pois todo o necessário já está implementado pelo framework. As classes de testes criadas com DUnit são bem simples, como deveria ser. Basicamente todos os testes unitários são criados com apenas dois passos: 1º) Você cria o ambiente; 2º) Você executa os testes.

Os testes unitários devem ser independentes entre si, isto significa que todos os testes devem ser capazes de rodar sozinho sem que outro teste precise ser executado antes. Além disto, os testes devem rodar sempre sobre as mesmas circunstâncias, o que significa que o teste não pode ser influenciado por variáveis externas como, por exemplo, um banco de dados bagunçado. Se for assim, um teste que funciona hoje porque o banco está "limpinho", pode não funcionar amanhã quando banco estiver povoado por outros testes: por isto o banco de dados deve sempre estar limpo quando o teste começar. Normalmente o próprio teste limpa o banco de dados antes de começar.

É comum que cada teste configure o ambiente antes de ser executado, como por exemplo, criar a conexão e limpar o banco de dados – as tarefas necessárias para criar o ambiente geralmente são repetitivas entre os testes. Para entender melhor o que significa configurar o ambiente de teste, imagine que você vai testar o CRUD (cadastro) de clientes. Cada cliente deve estar necessariamente ligado em uma empresa, logo para poder testar o CRUD de clientes é necessário que exista pelo menos uma empresa cadastrada, por este motivo, cadastrar pelo menos uma empresa é um exemplo de tarefa necessária para criar o ambiente de teste.

Depois de configurar o ambiente é hora de criar o teste propriamente dito: é isto que você verá a seguir.

## Praticando Desenvolvimento Ágil

Chegou o momento de juntar tudo que você viu até agora, para isto vou simular o desenvolvimento do CRUD de clientes. Os requisitos desta função foram definidos na **Listagem 2**. Com os requisitos definidos, resta apenas começar a codificar – como se trata de um projeto desenvolvido baseado em testes, a codificação sempre começa pelos testes e o objetivo do teste sempre é garantir que está sendo feito tudo que os requisitos mandam.

Na **Figura 1** está o diagrama de classes com as entidades necessárias para criar o CRUD. Com certeza, é possível criar um design melhor para esta aplicação, implementando padrões como Command ou Table Data Gateway. Isto provavelmente

tornaria o exemplo mais empolgante, porém seria necessário explicar muito sobre coisas que apesar de serem interessantes não são relevantes para o foco do artigo.

Para criar o CRUD de clientes somente duas classes serão criadas: TCliente e TClienteDao – todas as outras classes, como por exemplo, TEntidade, TEmpresa e TDao já existem no projeto. A classe TDao é uma classe abstrata que implementa as operações básicas para manipular uma tabela: Salvar, Buscar, Excluir e Listar. Além disto também declara (mas não implementa), dois outros métodos, são eles: AntesDeSalvar para validar as informações que serão salvas e AntesDeExcluir para verificar se um registro realmente pode ser excluído.

TDao não implementa estes métodos porque ela não pode prever todos os tipos de informações que serão salvas: TCliente, TEmpresa, entre outras. Mesmo que pudesse prever seria um caos testar tudo dentro da mesma classe, por isto TDao é abstrata. Sendo assim, deve existir uma subclasse de TDao para cada tipo de informação. Por exemplo, para manipular a tabela de cliente vai existir uma subclasse chamada TClienteDao e para a tabela de empresa vai existir TEmpresaDao. São as subclasses de TDao que implementam os métodos AntesDeSalvar e AntesDeExcluir.

A classe TDao sempre invoca o método AntesDeSalvar antes de salvar um registro e AntesDeExcluir antes de excluir um registro. Estes métodos por sua vez devem gerar uma exceção caso a respectiva operação não puder ser executada. Por exemplo, o método AntesDeSalvar da classe TClienteDao deve gerar uma exceção se o cliente estiver sem razão social.

Agora que você já está familiarizado com as classes do projeto, está na hora de implementar o CRUD de clientes, para isto primeiro os testes devem ser criados. Veja a seguir como criar uma classe de teste unitário com DUnit e entenda como ela funciona.

## Primeiro os testes

Todo teste é baseado puramente na lista de requisitos e foi lendo os requisitos da **Listagem 2** que descobri que será necessário criar duas novas classes: TCliente e TClienteDao. Como se trata de um projeto orientado a testes, os testes devem ser criados antes mesmo de criar as classes envolvidas no problema: por isto a primeira versão dos testes vai falhar obrigatoriamente.

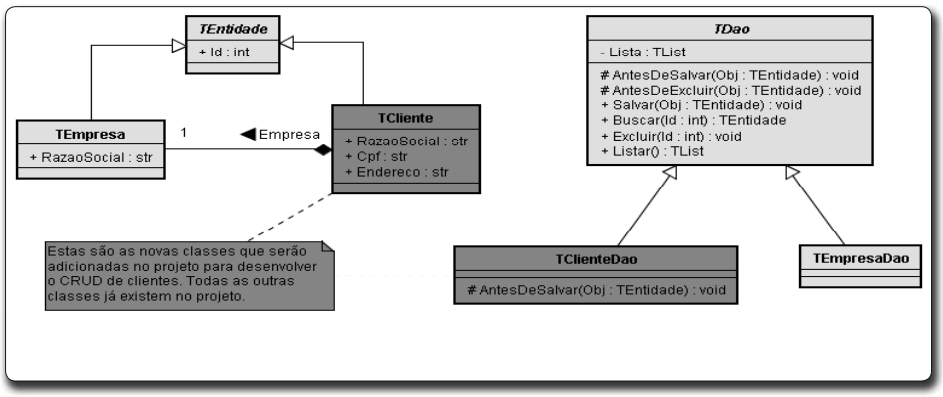



**Figura 1.** Classes envolvidas no problema "CRUD de clientes"

**Listagem 2**. Requisitos para o CRUD de clientes

1. O cliente deve estar ligado a uma empresa;
2. Não podem existir dois clientes com o mesmo CPF para a mesma empresa;
3. O usuário preenche apenas o nome e endereço do cliente.

**Listagem 3**. Exemplo de uma classe de teste vazia

```
01  unit TestClienteDao;
02  interface
03  uses
04   TestFramework;
05  type
06   TestTClienteDao = class(TTestCase)
07   published
08    procedure TestVazio();
09   end;
10  implementation
11  procedure TestTClienteDao.TestVazio;
12  begin
13   Fail('A classe  TClienteDao precisa ser
       criada');
14  end;
15  initialization
16  RegisterTest(TestTClienteDao.Suite);
17  end.
```

Mas como escrever um teste para uma classe que não existe? Existem duas alternativas. A primeira seria criar os testes e implantar uma falha propositalmente para indicar que as classes ainda não foram criadas (veja exemplo na **Listagem 3**). A segunda alternativa é criar as classes que serão testadas, mas não implementar nenhuma lógica, ou seja, você cria as classes e declara os métodos, mas não implementa nenhum deles. É desta forma que mostro a seguir.

Para começar vou criar as classes TClienteDao e TCliente, porém não vou implementar nenhuma lógica. Isso garante que o teste vai compilar, já que a classe TClienteDao e todos seus métodos foram declarados, porém o teste vai falhar, pois não implementei nenhum dos métodos da classe. Veja na **Listagem 4** a classe TCliente e na **Listagem 5** a classe TClienteDao.

Agora que as classes necessárias existem, é possível criar o teste. Para executar os testes é necessário que exista uma interface para mostrar o resultado dos testes executados. Esta interface já está pronta e para usá-la você só precisa criar um novo projeto específico para executar testes. Para criar um projeto de testes no Delphi 2010 você deve escolher a opção Test Project na janela New Items, como aparece na **Figura 2**. Ao clicar em "Ok", aparece um pequeno assistente com dois passos: no primeiro passo você poderá escolher o nome e o local onde o projeto de teste será salvo e no segundo passo apenas clique "Finish". Você pode salvar o projeto de teste em qualquer local, mas normalmente ele é salvo dentro de uma pasta chamada "Test" no diretório do projeto verdadeiro (aquele que será testado).

Ao executar o projeto de teste (criado no parágrafo anterior), o formulário da **Figura 3** vai aparecer. Esta é a interface do DUnit: é neste formulário que os testes são executados e os resultados serão exibidos. Você não precisa mudar ou configurar mais nada, tudo já está pronto, portanto, vou criar a primeira classe de teste unitário.

## Estendendo TTestCase

Para escrever os testes eu devo voltar aos requisitos (**Listagem 2**), e em uma folha de papel eu escrevo todos os testes necessários para garantir que nenhum requisito será esquecido. Veja na **Listagem 6** quais são os testes que eu encontrei para a lista de requisitos da **Listagem 2**.

Depois de definir os testes podem ser codificados. O Delphi possui um modelo para classes de testes e para usar este modelo basta criar um "Test Case", que aparece na **Figura 2**. Quando pedir para criar um "Test Case", o Delphi mostrará um assistente, com dois passos: no primeiro passo você escolhe a unit onde está a classe que deseja testar e no segundo passo você simplesmente clica em "Finish".

Neste exemplo o arquivo "ClienteDao.pas" será testado, pois o objetivo é testar a classe TClienteDao que existe dentro dele. Finalizando o assistente o Delphi mostra a nova a classe de teste TestTClienteDao – esta é a classe de teste unitário que vai testar a classe TClienteDao. Veja na **Listagem 7** como a classe TestTClienteDao aparece assim que o assistente para adicionar um "Test Case" é concluído.

Note na linha 6 que a classe TestTClienteDao estende de TTestCase. Esta é uma classe do framework DUnit – todos os seus testes unitários devem estender dela para poder funcionar. Na verdade esta classe faz muito mais do que simplesmente permitir que os testes funcionem. A classe TTestCase tem a maioria (se não todos) os métodos que você precisa para criar um teste.

Os dois primeiro métodos da classe TTestCase que você precisa conhecer são: SetUp e TearDown. O método SetUp é executado antes de cada teste e o método TearDown depois de cada teste – essa característica torna estes métodos ideais para configurar o ambiente de teste como é citado no tópico DUnit. Repare que o Delphi sugeriu uma implementação para este dois métodos: no SetUp é criada uma instância da classe TClienteDao (classe que será testada), e no TearDown a instância é destruída. Se fosse necessário criar alguma conexão com o banco ou fazer alguma

**Nota**

Toda classe deveria ter outra classe de testes específica para ela, mas neste exemplo eu criei um teste apenas para classe TClienteDao. Fiz isto porque a classe TCliente apenas declara atributos e não tem nada que precise ser testado. Na vida real, em um projeto mais robusto, também seria necessário criar uma classe de testes para TCliente.

**Listagem 4**. A classe TCliente

```
01  unit Cliente;
02  interface
03  uses Empresa, Entidade;
04  type
05    TCliente = class(TEntidade)
06    public
07      Empresa: TEmpresa;
08      RazaoSocial: string;
09      Cpf: string;
10      Endereco: string;
11    end;
12  implementation
13  end.
```

**Listagem 5**. O "esqueleto" da classe TClienteDao

```
01  unit ClienteDao;
02  interface
03  uses Dao;
04  type
05    TClienteDao = class(TDao)
06    end;
07  implementation
08  end.
```

**Listagem 6**. Testes para os requisitos da **Listagem 2**

```
01. Não pode existir um cliente sem empresa;
02. Não podem existir dois clientes com o mesmo CPF para
    a mesma empresa;
03. O campo razão social é obrigatório.
04. O campo endereço é obrigatório.
```

**Listagem 7**. Teste unitario para TClienteDao criada pelo Delphi

```
01  unit TestClienteDao;
02  interface
03  uses
04    TestFramework, Cliente, Dao, ClienteDao;
05  type
06    TestTClienteDao = class(TTestCase)
07    strict private
08      FClienteDao: TClienteDao;
09    public
10      procedure SetUp; override;
11      procedure TearDown; override;
12    end;
13  implementation
14  procedure TestTClienteDao.SetUp;
15  begin
16    FClienteDao := TClienteDao.Create;
17  end;
18  procedure TestTClienteDao.TearDown;
19  begin
20    FClienteDao.Free;
21    FClienteDao := nil;
22  end;
23  initialization
24    RegisterTest(TestTClienteDao.Suite);
25  end.
```

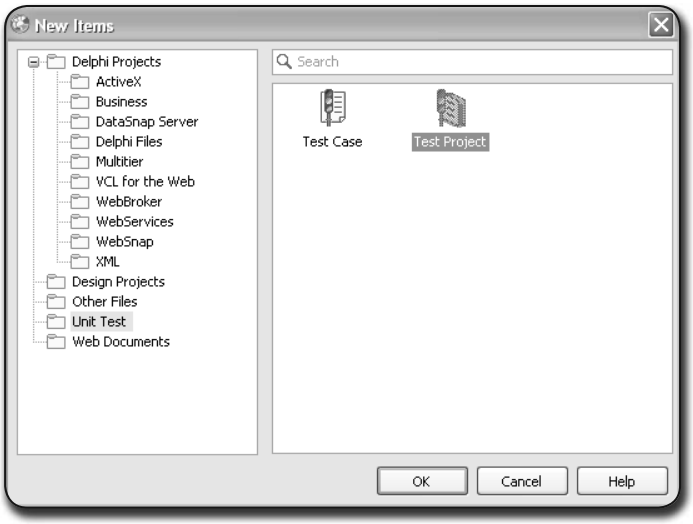



**Figura 2.** Criando um projeto de testes

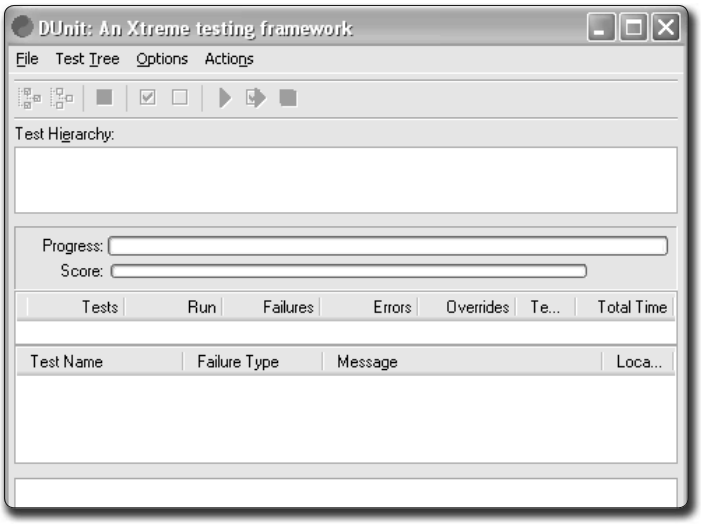



**Figura 3.** A interface do DUnit para executar testes

configuração extra antes de executar os testes, é neste método que estas ações devem ser executadas.

Para começar com a implementação dos testes eu vou configurar melhor o meu ambiente de teste, por isto, vou alterar a implementação do método SetUp (veja a nova implementação na **Listagem 8**). Assim como na implementação sugerida pelo Delphi, uma instância de TClienteDao continua sendo criada, mas eu adicionei algumas instruções mais interessantes. Veja na linha 4, que o método CheckEquals é executado para garantir não existe nenhum cliente cadastrado, pois isto poderia atrapalhar o teste.

O método CheckEquals é mais um dos métodos da classe TTestCase. Este método vai checar se o valor do primeiro parâmetro é igual ao valor do segundo parâmetro: se eles forem diferentes o teste falha – logo você verá como identificar se um teste falhou ou não. O terceiro parâmetro deste método – que é opcional – indica qual a mensagem que será exibida caso a checagem falhe, ou seja, caso o primeiro parâmetro seja diferente do segundo parâmetro. A classe TTestCase disponibiliza vários métodos que começam com "Check", cada um deles tem um objetivo diferente: dê uma olhada na classe TTestCase para ver todos os métodos de checagem existente – não são muitos.

Depois de instanciar TClienteDao e certificar de que não existem clientes cadastrados, é a vez de instanciar um objeto de TEmpresaDao e cadastrar uma empresa que será usada durante os testes. Mas uma vez uso CheckEquals, porém desta vez ele é usado para certificar que a nova empresa realmente foi cadastrada – note que agora o terceiro parâmetro não está sendo usado. O último detalhe antes de criar os testes é sobre o método NovaEmpresa. Cadastrar empresa é uma tarefa que pode ser repetida várias vezes ao longo dos testes e o método NovaEmpresa deve facilitar isto – repare que ele já foi usado uma vez na linha 6.

**Listagem 8**. O método SetUp definitivo

```
01  procedure TestTClienteDao.SetUp;
02  begin
03    FClienteDao := TClienteDao.Create;
04    CheckEquals(0, FClienteDao.Listar.Count,
      'Espera-se que a tabela de clientes esteja vazia.');
05    FEmpresaDao := TEmpresaDao.Create;
06    FEmpresa := NovaEmpresa(1, 'Empresa 1');
07    FEmpresaDao.Salvar(FEmpresa);
08    CheckEquals(1, FEmpresaDao.Listar.Count);
09  end;
10  function TestTClienteDao.NovaEmpresa
    (Id: Integer; Nome: string): TEmpresa;
11  begin
12    Result := TEmpresa.Create;
13    Result.Id := Id;
14    Result.RazaoSocial := Nome;
15  end;
```

## Revisão – como chegamos até aqui?

Primeiro houve a necessidade de criar o CRUD de clientes para um projeto que já existia. Os requisitos para este CRUD foram definidos e em seguida foi criado o esqueleto das classes necessárias para tal função. O "esqueleto" era as classes com a declaração dos métodos sem a implementação: o mínimo necessário para poder executar os testes sem criar erros de compilação.

A partir da lista de requisitos foi criada outra lista: a lista de testes. Nesta lista foram escritos todos os testes necessários para garantir que a classe TClienteDao funcione de acordo com os requisitos. Qualquer teste que não esteja validando um requisito é desperdício de tempo.

O segundo passo foi criar o projeto de testes. Normalmente, para um projeto em andamento o projeto de testes já estaria criado, mas fiz questão de explicar como criá-lo. Com o projeto de testes criado, foi adicionado um "Test Case" para a classe TClienteDao. Depois o ambiente de teste foi configurado modificando os métodos SetUp e TearDown.

**Listagem 9**. Seu primeiro método de teste

```
01  procedure TestTClienteDao.TestCampoRazaoSocialEhObrigatorio;
02  var
03   Cliente: TCliente;
04  begin
05   { Tenta salvar um cliente sem nome. }
06   Cliente := novoCliente(1, '', '11111111111', 'Rua X', FEmpresa);
07   try
08    FClienteDao.Salvar(Cliente);
09   except
10   end;
11   CheckEquals(0, FClienteDao.Listar.Count, 'Salvou cliente sem nome');
12   { Salva cliente com um nome válido }
13   Cliente := novoCliente(1, 'Miguel', '11111111111', 'Rua X', FEmpresa);
14   FClienteDao.Salvar(Cliente);
15   CheckEquals(1, FClienteDao.Listar.Count, 'Não salvou cliente com nome válido.');
16  end;
17  function TestTClienteDao.NovoCliente(Id: Integer; Nome, Cpf, Endereco: string;
18   Empresa: TEmpresa): TCliente;
19  begin
20   Result := TCliente.Create;
21   Result.Id := Id;
22   Result.RazaoSocial := Nome;
23   Result.Cpf := Cpf;
24   Result.Endereco := Endereco;
25   Result.Empresa := Empresa;
26  end;
```

Agora vem a parte mais interessante: criar os métodos de testes propriamente dito. Existirão tantos métodos de teste quantos foram os testes definidos para validar os requisitos. Veja a seguir como criar e executar os testes.

## Criando métodos de teste

De acordo com a **Listagem 6** devem existir quatro métodos de testes. Veja na **Listagem 9** como o teste número 3 da **Listagem 6** foi transformando em método de teste para a classe TestTClienteDao: foi criado o método "TestCampoRazaoSocialEhObrigatorio". É recomendado que os testes tenham um nome que deixe bem claro o que ele está testando, por isto um nome tão grande.

Os métodos de testes são bem simples de entender: você executa alguma coisa e em seguida executa um método "Check" para garantir que o que você executou realmente funcionou. Por exemplo, entre as linhas 6 e 10 tenta-se cadastrar um cliente sem razão social. O comportamento esperado é que o cliente não seja cadastrado, por isto na linha 11 um CheckEquals é executado para garantir que nenhum cliente foi cadastrado.

Este primeiro teste tenta cadastrar um cliente sem razão social e isto provavelmente deve gerar uma exceção. Por isto o try..except entre as linhas 7 e 10. Este try.. except esconde qualquer exceção que possivelmente vai existir quando um cliente sem razão social tentar ser cadastrado.

Depois de garantir que um cliente sem razão social não pode ser cadastrado, chegou a hora de garantir que um cliente com razão social poderá. Isto acontece nas linhas 13 e 14 – note que desta vez não foi usado try..except. Na linha 15 um novo CheckEquals é executado para garantir que existe um novo cliente cadastrado.

Antes de executar o teste, é necessário destacar que agora existem dois projetos que funcionam em paralelo: um projeto é a aplicação de verdade e outro projeto serve apenas para executar os teste. Este segundo projeto é aquele que expliquei como criar no tópico "Primeiro os testes". Uma vez que existem dois projetos, você deve adicionar manualmente no projeto de teste todas as units necessárias para fazê-lo compilar. Neste exemplo, foi necessário adicionar as units TDao, TEmpresaDao, TEmpresa e TEntidade (veja diagrama na **Figura 1**). As classes TClienteDao e TCliente foram adicionadas automaticamente quando criei a "Test Case" no tópico "Estendendo TTestCase".

Feita esta observação, você está ponto para saber que só precisa pressionar F9 para executar a aplicação de testes e a partir dela, executar o teste criado na **Listagem 9**. Veja na **Figura 4** a imagem da aplicação de teste após executar os testes. Existem algumas observações importantes sobre as informações que aparecem no formulário da **Figura 4**.

A primeira coisa é que somente o método "TestCampoRazaoSocialEhObrigatorio" foi reconhecido como sendo um método de teste. Os outros métodos como, por exemplo, "NovaEmpresa" e "NovoCliente" não foram. Isto acontece por que os métodos de teste devem estar declarados na sessão published. Todos os métodos nesta sessão serão reconhecidos como métodos de testes.

Outra informação importante é que todos os testes falharam (neste caso só existe um). Mas isto já era esperado, pois ainda não existe nenhuma lógica na classe TClienteDao. Repare que a mensagem "Salvou cliente sem nome" aparece na parte destacada da imagem. Para resolver este problema eu devo ir até a classe TClienteDao e implementar o método AntesDeSalvar para testar se o cliente possui ou não possui razão social. Veja esta alteração na **Listagem 10**.

Depois de alterar a classe TClienteDao, o teste deve passar porém, antes de executá-lo mais uma vez para garantir isto, eu vou declarar os demais métodos de teste definidos na **Listagem 6**. Veja a declaração destes métodos na **Listagem 11**. Para poupar espaço eu não vou implementar estes métodos de teste, vou apenas fazer a declaração e forçar a falha deles com uma mensagem indicando que eles ainda não foram implementados. Agora ao executar os testes novamente, o resultado será o que aparece na **Figura 5**.

É possível notar nesta figura que a alteração feita na classe TClienteDao teve um efeito positivo, pois o teste "TestCampoRazaoSocialEhObrigatorio" parou de falhar. Por outro lado os demais testes estão falhando propositalmente porque ainda não foram implementados. Implementar estes métodos é um bom exercício para colocar em prática o que este artigo ensina: baixe

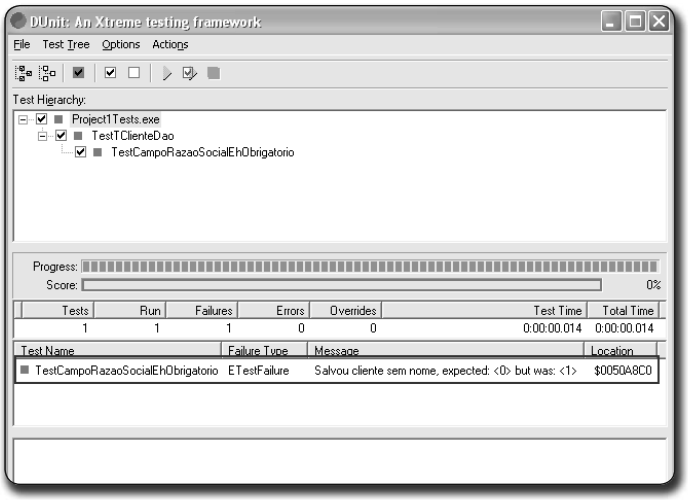



**Figura 4.** Executando os testes

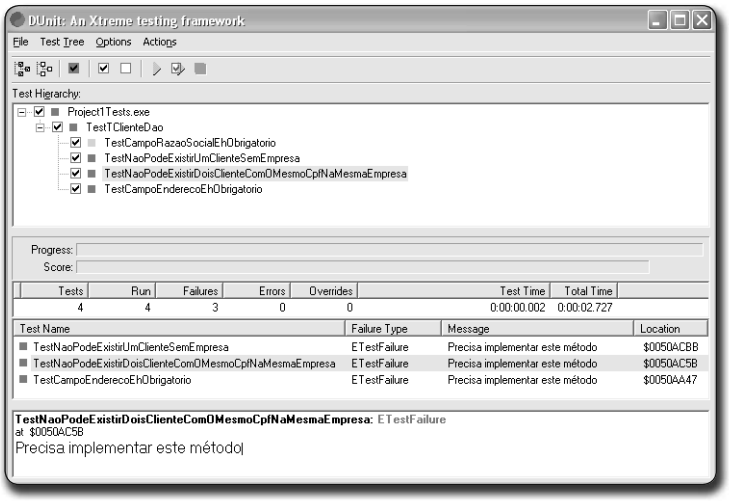



**Figura 5.** Executando os testes novamente

o arquivo deste artigo e implemente os métodos de testes que estão falhando.

## Cobertura de código

Cobertura de código é uma técnica usada para verificar se os testes que você criou deixaram alguma coisa sem testar para trás, ou seja, é uma técnica para descobrir quantas linhas de códigos não foram cobertas pelos testes escritos. Discover é um programa open-source usado para medir quantas linhas de código faltam ser cobertas. Ele não precisa ser instalado, basta executá-lo para poder usar. O link para a página de download do Discover está na sessão Links do artigo.

Para usar o Discover você precisa do arquivo de detalhamento do projeto cujo percentual de cobertura será calculado. Este arquivo é criado quando a opção 'Map file' do Delphi está habilitada. Para configurar o Delphi para criar o arquivo de detalhamento você deve abrir o projeto no Delphi e dentro de "Project + Options" você deve escolher "Detailed" para o item "Map file" da sessão "Linking" – Veja na **Figura 6**. Esta configuração deve ser feita no projeto de testes e não no projeto verdadeiro.

"Depois de configurar o "Map File", será necessário executar o projeto uma vez para que o arquivo".map" seja criado. Após executá-lo, o Delphi não será mais necessário e se preferir poderá fechá-lo. Agora o trabalho restante é feito no Discover. Para descobrir o percentual de código ainda não coberto pelos testes, execute o Discover, clique no menu "Project + Load" e selecione o arquivo ".dpr" do projeto de testes (aquele que foi configurado o arquivo ".map").

Depois de carregar o projeto para o Discover, já é possível executar a aplicação escolhendo o menu "Application + Run" do Discover. Quando o projeto de testes for iniciar basta excutá-lo normalmente como foi feito no tópico "Criando métodos de teste". Depois de executar os testes o Discover mostrará o percentual de código coberto. A seguir você vai aprender a ler as informações que o Discover gerou.

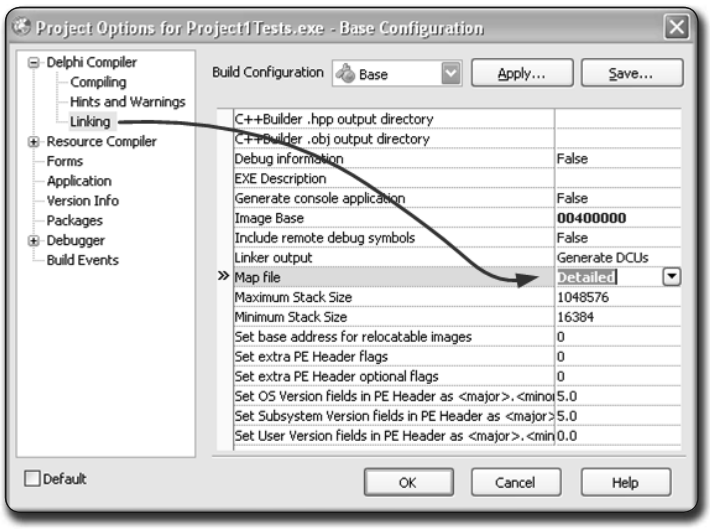



**Figura 6.** Configurando o Delphi para gerar o arquivo ".map"

**Listagem 10**. Alterando TClienteDao para fazer o teste passar

```
01  procedure TClienteDao.AntesDeSalvar(Obj: TEntidade);
02  begin
03    if Trim((Obj as TCliente).RazaoSocial) = '' then
04      raise Exception.Create('o nome do cliente é obrigatório.');
05  end;
```

**Listagem 11**. Os demais métodos de teste

```
01  procedure TestTClienteDao.TestNaoPodeExistirDoisCliente
      ComOMesmoCpfNaMesmaEmpresa;
02  begin
03    Fail('Precisa implementar este método');
04  end;
05  procedure TestTClienteDao.TestNaoPodeExistirUmCliente
      SemEmpresa;
06  begin
07    Fail('Precisa implementar este método');
08  end;
09  procedure TestTClienteDao.TestCampoEnderecoEhObrigatorio;
10  begin
11    Fail('Precisa implementar este método');
12  end;
```

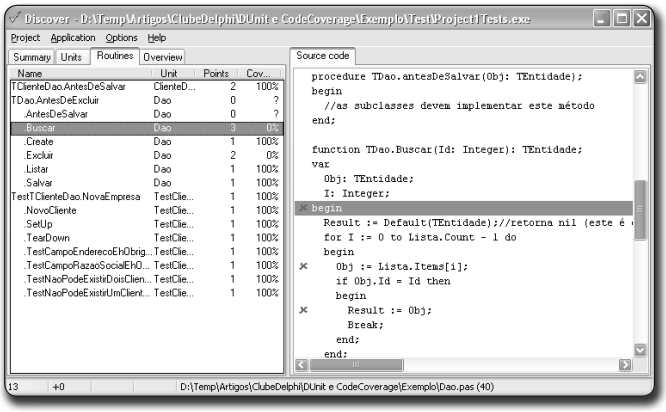



**Figura 7.** A guia Routines do Discover

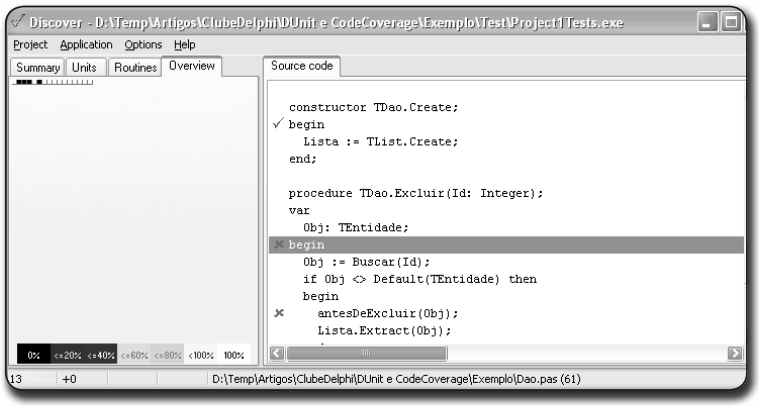



**Figura 8.** A guia Overview do Discover

Dentro da guia Routines (**Figura 7**) do Discover, estará o nome de todas as rotinas do programa e o percentual de cobertura de cada uma – o percentual de cobertura aparece somente depois que você executar a aplicação, como expliquei nos dois últimos parágrafos. Ao clicar no nome de uma das rotinas será exibido do lado direito o código fonte, com um "X" ao lado da linha que não foi executado ainda.

A guia Overview (**Figura 8**) apresenta outro ponto de vista do código. Esta guia mostra a aplicação de um ponto de vista global. Todas as rotinas são representadas por um pequeno quadrado que aparece no painel esquerdo. Cada quadrado terá uma cor indicando o percentual de cobertura: o valor percentual que cada cor representa aparece na parte inferir do painel esquerdo. Clicando em um dos guardados, aparece no painel direito a parte do código fonte que ele representa.

## Conclusão

"Testar" é parte do desenvolvimento de software tão importante quanto "codificar". As técnicas apresentadas neste artigo são apenas algumas das existentes. Para aqueles que pretendem se aprofundar na parte teórica do assunto eu recomendo a leitura de um bom livro para ter conhecimento de todas as demais técnicas e fases existentes.

**Rafael Stavarengo**
*faelsta@gmail.com*
*http://stavarengo.com*
Programador de sistemas com 8 anos de experiência, integrante da equipe editorial da revista ClubeDelphi. Domínio em Java, PHP e UML. Sólido conhecimento em Design Patterns e metodologias ágeis. Graduado em Análise e Desenvolvimento de Sistemas pela UNIPAR.

**The Agile Alliance**
*http://www.agilealliance.org*

**Download do Discover**
*http://sourceforge.net/projects/discoverd*

**Dê seu feedback sobre esta edição!**

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

***www.devmedia.com.br/clubedelphi/feedback***

# RENOVE JÁ!

## Sua assinatura pode estar acabando

**Renovando a assinatura de sua revista favorita você ganha brindes e descontos exclusivos.**

---

**Faça a renovação de sua assinatura agora mesmo.**

**www.devmedia.com.br/renovacao ou (21)3382-5038**

DevMedia

# Hospede seus sites de forma simples, rápida e ilimitada.

## Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

**HOSPEDIX**
HOSPEDAGEM PROFISSIONAL
**www.hospedix.com.br**

**30 DIAS GRÁTIS**

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

**RVSTMEDIA**

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.